Constructive Semantics

by

Paul Cameron Brown

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved by the
Examining Committee:

Mukkai S. Krishnamoorthy, Thesis Advisor	David R. Musser, Thesis Advisor
David L. Spooner, Member	Edward H. Rogers, Member
Philip M. Lewis, Member	Kim B. Bruce, Member

Rensselaer Polytechnic Institute

Troy,New York

May 1992

Revised 1/28/99

Contents

List of Figures

Acknowledgments

It is difficult to know where to begin in acknowledging the people who have contributed, directly or indirectly, to this work. I suppose that an appropriate place to begin is with the teachers who have encouraged and inspired me to go beyond rote learning in a very fundamental way and delight in truly *understanding* things: Richard Kimball, Cornelius Banta, Dick Russ and Bill Stone. My thanks also to those who were my guides as I began to understand this new and exciting field of computer science: Mukkai “Moorthy” Krishnamoorthy, Dave Spooner, Deepak Kapur, Bob McNaughton, Sam Kim and Erich Kaltofen.

In exploring a field, there comes a point when one begins to ask questions for which the answers are not readily found, and one begins to experiment with new ideas as a means of arriving at the answers to these elusive questions. These initial steps toward research are tenuous at best: ideas frequently turn out not to be new at all, and those that are new require enormous effort to develop into clearly defined concepts that can be communicated and used.

It is the difficult task of the advisor to guide the emerging researcher through this troubling period. My first advisor was David Spooner, who helped me through the first tentative steps back when I was completing my Master’s Degree. As my interest shifted toward semantics, “Moorthy” Krishnamoorthy became my advisor for my doctoral work, and was later joined by Dave Musser as the work expanded to encompass the Ada programming language. Dan Rosenkrantz and Phil Lewis have also helped me through this process of learning to do research. I have much to thank these people for: the freedom to both define the problem area and carve out the approach to the solution; the hours that they spent listening patiently to half-developed ideas as I tried to understand them myself; and perhaps most of all, for not discouraging me by pointing out the magnitude of the task that I had set out for myself.

A special thanks to Robin Milner, whose work has laid the foundation for this thesis, for helping me understand why I had been unable to prove the parallelization theorem using ASCCS and observational equivalence, and to Phil Lewis, who first pointed out Milner's work to me and kept me from re-inventing a rather large wheel.

This thesis was inspired by some work that I did on a project at GE's Corporate Research and Development Center developing an experimental incremental compiler and development environment for the Ada programming language. During the course of that project I conceived many of the ideas central to this thesis, although it has taken me a long time to evolve them into the form in which they presently appear. My thanks go to Phil Lewis and Joel Sturman for the opportunity to work on the semantic model for that project, and to Dave Oliver and Art Chen for their continuing support as the project dragged on. Jim Guilford, Tim Kelliher and Alyce Stewart struggled through applying the partially developed ideas in the incremental compiler, and provided valuable feedback that helped the model evolve, while John Hutchison provided a running sanity check on my understanding of Ada.

This work has not come without a price. My wife, Maria, has been waiting patiently for 16 years as I have worked my way through three degrees while also working full time. During that time she has tirelessly done many of the things that I should have been doing myself so that I could have the time to study. Without her continuing love and support, this work would never have been accomplished. My daughter Jessica, now in High School, has grown up watching me study all the time. My son Philip, now in first grade, has been counting the days until this thesis was completed so that we might play. Well the waiting is over...it's playtime!

Abstract

Constructive Semantics is an approach to programming language semantics that treats a program as a constructive specification for an abstract state machine. This abstract machine is composed of a set of smaller "well-behaved" machines operating concurrently. The exact combination of machines is determined by the program, with each programming language construct appearing in the program defining a portion of the composition. The programming language itself specifies a number of primitive machines that form the basic building blocks of programs. These machines represent the basic operations and data types of the language. The resulting semantics is relatively easy to understand, an its relationship to the original program is clear.

Constructive semantics treats many higher level programming language abstractions also as specifications of state machines, where these machines serve as prototypes for entire sets of machines. For example, a basic data type in a programming language is modeled as a state machine, and each variable of the type is modeled as a copy of this machine. Behavioral equivalence of machines provides a basis for modeling abstract data types, in which behaviorally equivalent machines belong to the same abstract data type. Behavioral equivalence also provides a basis for modeling type hierarchies such as those found in object-oriented languages with multiple inheritance.

Ada generics and C++ templates are modeled as partial specifications of state machines. These partial specifications contain variables corresponding to the formal parameters of the generic or template. The expected values for these variables are state machines. An instantiation of the generic or template is modeled as the state machine defined by replacing each variable with the state machine corresponding to the actual parameter (usually the prototype machine associated with a type or subprogram).

Constructive semantics provides straightforward semantic models for other important aspects of programming languages, including concurrency (Ada tasks), elaboration and visibility computations. Several theorems in constructive semantics describe orthogonality (independence) conditions under which the serial/parallel relationships between machines in a composition may be modified without affecting observable behavior. The formalism underlying constructive semantics is derived from the well-studied models of Milner's Calculus of Communicating Systems (CCS) and Hoare's Communicating Sequential Processes (CSP). Behavioral equivalence in constructive semantics is based upon Hoare's concept of failures equivalence.

1.0 Introduction and Historical Review

Semantics is "The relation between signs or symbols and what they signify or denote.[1]" For most programming languages, the rules for identifying the signs and symbols (the lexical elements of the language) are well defined and easily understood, as are the rules for defining the legal structures of these elements (the syntax). Less well defined are the *concepts* that are *denoted* by the elements of the language and the *relationship* between the symbols and the concepts. These concepts, and the relationship that they bear to the symbols of the language, are the subject of this thesis.

To set the stage for this work, we must ask two questions: What is the purpose of semantics? and Who are the intended users of semantics? Presumably the intent of semantics is to describe the meaning of a program in a way that is, in some sense, more understandable than the program itself. The driver for increased understanding may be the need for more precision (finer detail), or the need to better define an abstract concept ("...just what *is* a type, anyway?") In either case, the semantic model must be *understandable* to be of practical value.

Who are the intended users of semantics? Ultimately, we would like the audience to be the body of language designers, complier and interpreter writers, and software engineers that will work with the language whose semantics is being defined. This places great demands upon the semantics. Providing an elegant mathematical model in category theory, for example, is not going to help the average software engineer understand the language better. On the other hand, describing semantics with readily understood but somewhat informal models is not going to provide the precision necessary to illuminate the nooks and crannies of the language and clarify what otherwise might be ambiguities.

1. Funk & Wagnalls Standard Dictionary of the English Language, International Edition (1962)

1.1 Current Approaches to Semantics

There are four primary approaches to semantics commonly found in the literature today, differing mainly in the abstractions that they use to model a program. *Denotational semantics* takes a functional approach, translating program elements into lambda calculus expressions, and thus into the abstract mathematical domains of *values* and *functions*. *Algebraic semantics* is similar to denotational semantics, but translates programs into abstract algebras instead of the lambda calculus. *Axiomatic semantics* has a slightly different flavor, giving assertions about the behavior of program elements in axiomatic form. *Operational semantics* maps programs onto the behavior of some abstract machine.

To date, none of these approaches has yet yielded (to the best of our knowledge) a complete semantics for a strongly typed production programming language. Even those that have been partially completed are difficult to understand [Modu88][Bjor80]. We believe that this lack of success does not necessarily indicate that the models being used are fundamentally inappropriate for the task at hand. We believe that the real problem is that a higher level of abstraction, which is in turn built upon one or more of these models, is necessary to bridge the gap between the model and the programming language and clarify the relationship between the program and the model. Before expanding upon this concept, let us briefly examine the four popular models.

1.1.1 Denotational Semantics

Denotational semantics translates program elements into lambda calculus expressions, thus describing the meaning of the program in terms of the abstract mathematical domains of values and mappings (functions) between the values [Scot76][Scot82][Stoy77]. Semantics for more elaborate languages treat types and functions themselves as values in this domain,

and add appropriate mappings to represent parameterized types and functions [Amad86] [Bruc80][Bruc84][McCr82][Mitc84].

Since the denotational model is functional in nature, the resulting semantic expressions typically take two fairly complex values as arguments and return these as secondary values (possibly in modified form) along with the primary value being returned from the expression. These complex values are the state of the "memory" or *store*, and "the rest of the program" or *continuation*. In general, a semantic expression may use and/or modify both the store and the continuation in the process of computing the value that it returns.

This passing of the store and continuation through virtually all semantic expressions makes it difficult to understand the scope of effect of changes in these variables. While it may be obvious from a local examination of a particular semantic expression whether it uses and/or modifies the store, it is *not* obvious from a local examination whether *the rest of the program* will use or modify these same values. Since there is no *explicit* indication of the possible interactions between any two semantic expressions, the effect of each expression upon the rest of the program is only partially specified by the expression itself. Assessing the relative dependence or independence of semantic expressions thus requires an analysis of the expression continuations and stores at every point in the program where the expressions in question are used.

Unfortunately, software engineers do not normally think of the modules of a program as pure functions. In fact, programmers tend to view modules as *machines with state* that respond to inputs by *changing state* and/or *producing outputs*. It is our belief that denotational semantics has not been a more successful vehicle for communicating the semantics of programming languages precisely because information concerning the scope of effect of

state changes and the possible interactions between semantic model elements is difficult to extract from the model.

1.1.2 Algebraic Semantics

Algebraic semantics [Broy87][Gogu77][Moss83][Moss84], like denotational semantics, seeks to map programs onto mathematical domains, with the distinction that in algebraic semantics multiple domains (algebras) are being used. Frequently the program itself is interpreted as defining and then using one or more algebras. Unfortunately, algebraic semantics, like denotational semantics, has a difficult time representing state. Variables, for example, are represented as constants in an algebra characterizing the data type. Changing the value of the variable creates a new algebra that is identical to the previous algebra except for the value associated with the constant. As with denotational semantics, we believe that this will limit the usefulness of algebraic semantics as a communications medium between language designers, compiler writers and software engineers.

1.1.3 Axiomatic Semantics

Axiomatic semantics [Hoar69][Mann74][Ende73][Mend87] brings the power of mathematical logic to bear upon programs and programming languages. Unlike the other semantic approaches, axiomatic semantics is not constructive, which requires the user of the semantics to view the program from a slightly different perspective.

Axiomatic semantics characterizes a program or program element by constraint: each axiom specifies some relationship that the program must be faithful to. In using axiomatic semantics, one is continually faced with the question of sufficiency with respect to the set of axioms provided. If the desire is simply to specify some minimum set of properties that a program or element must meet, then any consistent set of axioms is sufficient. If, on the

other hand, the desire is to *completely* specify a program to the extent that any two programs that are consistent with the axioms are guaranteed to behave in exactly the same manner under all circumstances, then a notion of equivalence is required. We suggest that Hoare's failures equivalence,[1] which we shall also take as our notion of behavioral equivalence in this work, offers a good definition of equivalence for this purpose, and also provides a basis upon which to determine whether a given set of axioms is complete. Furthermore, as we shall see in the thesis itself, the notion of *abstract* type that we will develop is also based upon this equivalence, thus providing a formal link between our model and this stronger form of axiomatic semantics.

1.1.4 Operational Semantics

Operational semantics models a program by mapping program elements onto the operations of an abstract machine, thus defining the behavior of the program in terms of the behavior of this abstract machine. In early operational semantics, this machine was often a relatively simple abstract computational engine whose behavior was defined by a set of rules.The selection of a particular abstract machine to be used as the basis for the semantics was frequently motivated as much by the availability of a well understood abstract machine as it was by considering the appropriateness of the abstract machine for modeling a particular language. Although this early form of operational semantics provided an explicit model for the notion of program state, the mapping from the state of higher level programming abstractions onto the state of the an abstract machine (like a stack machine) was often complex, thus limiting the understandability of such models. Additionally, it has been unclear how to model higher level program abstractions such as data types in operational semantics.

1. [Hoar85] p. 130, axioms C0-C3. Axioms C4-C6 extend failures equivalence to consider divergence (infinite computations) as well. We have left the issue of divergence open in this work.

More recently, operational semantics has taken a theoretical turn. Milner [Miln80][Miln83][Miln89] and Hoare [Hoar85] have both proposed calculi for *defining* abstract machines with well understood properties and operators. With this capability in hand, one can now entertain the thought of *customizing* the abstract machine to suit the language.

In this thesis we will carry this thought through to its logical conclusion: the program itself will be interpreted as the *definition* of an abstract machine. Milner[1] has explored some of the possibilities in this area, using CCS to model a simple imperative language with concurrently executing subprograms and shared variables. This work left open the question of how to model call-by-reference in subprogram calls, and did not address at all the area of types and type checking.

1.2 Contributions

We take a constructive approach to semantics in which the entire program is considered to be the *definition* of an abstract state machine and its initial state. We call this approach ***constructive semantics***. We have carried the earlier work of Milner[2] forward into a more general formulation capable of modeling types, type hierarchies, type checking, visibility computations and overload resolution.

To give a formalism to our semantics, we define a language for specifying state machines that we call a ***Hybrid Calculus of Communicating Systems*** (***HCCS***), since it borrows heavily from Milner's CCS and Hoare's CSP. We show that Hoare's failures equivalence is a congruence relation in this calculus, and define this to be our notion of behavioral equivalence.

1. [Miln89] pp. 170-185.
2. Ibid.

We define ***state types*** to be constructively specified sets of isomorphic state machines, and ***abstract types*** to be constructively specified sets of behaviorally equivalent machines. This behavioral equivalence then defines a partial ordering on types, and provides a formal basis for type hierarchies and the more complex type schemes of multiple inheritance. We define ***generic types*** to be parameterized specifications for state types, where the parameters are, themselves, other state machines.

Rather than translating a program directly into HCCS, we define an intermediate language that we call a ***Machine Algebra***. This offers considerable notational simplification over HCCS, since a single term in the machine algebra corresponds to both a term and a set of defining equations in HCCS. We show that by restricting the use of HCCS in defining machines, we can create a class of ***well-behaved*** machines whose starting and stopping behavior is well defined. We show that this property is preserved by the combinators of our machine algebra. We also show how restricting the use of some actions to guarantee that they only appear on one machine can be used to structure entire classes of machines that are, by construction, independent of one another. We define the ***orthogonality*** relation of machines that cannot interact with each other.

The usefulness of our machine algebra approach is demonstrated through two theorems, the ***parallelization theorem*** and the ***change of scope theorem***, each stating a set of orthogonality conditions under which a change in the serial/parallel relationships between machines in a machine algebra expression may be altered without affecting the observable behavior of the state algebra expression.

We review the notion of visibility in programming languages. We provide a formal definition of the notion of ***homograph***, and propose a modified set union operation that we call a ***masking union*** for use as a fundamental operation in defining the visibility of program

elements. We demonstrate the use of the masking union in modeling a wide variety of different visibility semantics.

Finally, we show how to use machine algebra and HCCS to give the constructive semantics for a programming language. We give a formal definition of ***elaboration*** as the process of converting a program into a state machine. We reconcile three seemingly disparate notions of type, giving a common model based on state types. We show that programs, subprograms and type declarations all define state types, thus providing a very general and uniform model for these apparently disparate concepts. We show that the *elaboration* of a variable declaration and the *calling* of a subprogram both cause instances of the state type to be created. We demonstrate, in a limited form, the use of generic machines.

Our model has a number of advantages over other semantic approaches. First, it is completely *language independent*, and thus provides a precise neutral ground for comparing and contrasting languages and language features. Second, while we do not have a normal form for expressions, we claim that our semantics is *fully abstract* in the sense that the equivalence classes of semantic expressions under behavioral equivalence *are* the fully abstracted semantics. Third, our semantics provides a uniform treatment of subprogram calls regardless of whether the subprogram is a function or a procedure, and regardless of whether the call occurs as a statement or as part of an expression. Finally, our semantics provides a domain for constructing accurate models of implementation strategies and formally comparing them (via behavioral equivalence) to an abstract semantics of a language, with the added advantage that these comparisons can be meaningfully carried out on fragments of the language.

1.3 Thesis Outline

Chapter 2 lays the groundwork for our semantics. In this chapter we define HCCS, a language of states and transitions used for describing state transition systems, which we will use as means of defining state machines. We explore the mechanism for interaction *between* state machines via the actions of the machines. We define a *behavioral equivalence* between machines, and show that this equivalence is a congruence relation in HCCS. We define the *orthogonality* of two machines to be the inability of the machines to interact with each other. We define *state types* to be sets of state machines that are isomorphic under an operator mapping the actions of one machine onto the actions of another machine. We define *abstract types* to be sets of machines that are behaviorally equivalent under a similar mapping. Finally, we introduce *reduced Petri nets* as a convenient means of illustrating interactions between machines.

HCCS is capable of defining a very broad class of machine, much broader than we wish to use in our semantics. In chapter 3 we turn to defining the constraints that we will impose upon the machines that are definable in constructive semantics, and raise our level of abstraction, focusing on entire machines and their relationships with one another. We define well-behaved machines to be machines with well defined initialization and termination properties.We define a *machine algebra*, a notation for describing the serial/parallel relationships between machines. This algebra will form the notational basis for our semantics. We define several basic classes of machines: *value machines*, which are mutually orthogonal machines intended to store values; *interaction machines* transfer values between machines, possibly computing values in the process; and *activation machines* serve to coordinate the starting and stopping of machines. We give an informal example of how these three classes of machine can be used to give a semantics to a block of code containing declarations and statements. We observe that the resulting expression appears to be, at first glance,

arbitrary. We then give two theorems that state orthogonality conditions under which machine algebra expressions can be transformed into other behaviorally equivalent expressions. We conclude the chapter by using one of these theorems to show that two apparently different semantics for an example program are in fact behaviorally equivalent.

In chapter 4 we turn to the topic of establishing the relationship between the identifiers in a program with the machines that they represent. We define an *environment* to be a mapping from identifiers to machines, and quickly see that many environments (many different mappings) are used in resolving the meanings of identifiers in programming languages. We define a *declaration* to be an association of an identifier with a machine, and a *reference* to be the occurrence of an identifier whose association with a machine must be determined through the examination of declarations in a particular environment We formalize the notion that certain pairs of declarations maybe indistinguishable in the form of a *homograph relation*, and discuss the differing definitions of homograph that arise in different programming languages. We then define a modified set union operator known as a *masking union*, and show how this operator can be used to model the computation of the contents of environments in terms of other environments. These computations determine the visibility of a given declaration at each point in the program. We explore three different visibility computations used in Lisp and Scheme, and discuss the modeling of some of the more complex Ada visibility rules, including the selected component or “dot” notation.

In chapter 5 we put the results of the previous three chapters to work and show how they are used to specify the constructive semantics of a programming language. There are three basic concepts that underlie our semantic model: first, an *executing program* is simply a state machine; second, a *program* is a *specification* for a state machine. Third, a *programming language* is simply a *language for specifying* state machines. Our approach to semantics will be to give the semantics of a program not in terms of a concrete state machine (i.e.

the computer itself), but rather in terms of an abstract state machine. Our goal is to make this abstract state machine both precise and easy to understand so that it may serve as an aid to the practicing programmer and the compiler/interpreter writer.

Obviously giving the semantics of a program as a single giant state machine would be of little benefit to either the programmer or the compiler writer. Consequently, just as the computer itself is a composition of state machines (ALU, registers, busses, memory, etc.) our abstract machine will also be a composition of smaller machines. These component machines are either primitive machines given by the language itself, or other machines that have been specified elsewhere in the program.

We shall see that a programming language can be viewed as two components: the specification of a small number of primitive machines with known properties (the basic data types and operations of the language); and the specification of a syntax for indicating how machines are composed to form larger machines. Most languages provide a syntax for first defining new machines and then using them as components of other machines. These syntax forms are used for user-defined data types, subprograms, packages, tasks and other high-level abstractions that can be specified in the language.

From the compiler or interpreter writer's perspective, any behaviorally equivalent state machine is then a valid implementation of the program. The challenge for them is to establish the behavioral equivalence of the implementation with the abstract machine given in the constructive semantics. Here the properties of composition come into play: it is sufficient to separately establish that each of the component pieces is equivalent and that each composition operation is equivalent, since all of the compositions that we shall use preserve behavioral equivalence.

Perhaps the major difference between our abstract semantics and a concrete implementation is that the abstract machines will not necessarily be finite in number. For example, in our semantics, a recursive function call will result in a theoretically infinite chain of (isomorphic) machines, each one implementing the function, and each one interacting with its caller (if any) and the machine that it, in turn, calls (we shall see later how information can be shared between invocations). This leads to a very straightforward model that is easy to understand and to reason about. However, in actual implementation, the compiler writer will not want to duplicate the entire machine for each invocation: typically only the data and the program counter will require additional storage for each invocation. It is then the compiler writer's task to satisfy himself that this implementation is behaviorally equivalent to the abstract semantics.

The term type frequently brings to mind differing and possibly inconsistent concepts. The three dominant concepts seem to be that a type is either a set of values[1], a behavioral (interface) specification, or an implementation specification. We show that state types and abstract types together provide a uniform framework for modeling all three concepts.

Not surprisingly, an executing program or subprogram is modeled as a state machine. Here we must be careful to distinguish between the *executing machine* itself and its *definition*, namely the program or subprogram itself. The *executing program* is an actual state machine, similar in nature to a variable. The *definition* is a specification of a class of machines, each one of which executes the program or subprogram. Programs and subprograms are thus definitions of state types: classes of isomorphic state machines. We shall refer to the executing machines as *instances* of the program or subprogram.

1. Or a representative of the set, as is the case in Denotational Semantics.

We formalize constructive semantics in the following way. We formalize the notion of *elaboration* as the process of converting a program into a state machine. We explore three common concepts of type, showing that they are all subsumed by state types and abstract types. We then give a formal semantics for a number of programming language constructs taken from Ada: variables, pointers, subprogram calls and expression evaluation, go-to statements, exception handling, loops, declarative blocks, record and array declarations, and program and subprogram declarations. These semantics are given in a style similar to that of denotational semantics, giving the meaning of each expression as a composition of the meanings of the sub-expressions. We explore the semantics of declarative blocks, noting how visibility computations affect the form of the semantic expressions. Finally we explore programs and subprograms, looking at the various alternatives for parameter passing.

Chapter 6 summarizes the current work, drawing some conclusions about how the current work could be applied and outlining some possibilities for future work.

The appendices contain the details of the mathematical formalisms underlying our work. Appendix A contains the formal definition of HCCS. Appendix B contains the proofs of several theorems about machine algebra expressions. Appendix C contains a brief summary masking union properties and the proof of one theorem regarding visibility computations.

2.0 State Machines and Machine Types

In this chapter we define state machines as they will be used in this thesis. In the process, we will actually be working with state machines on two different levels of abstraction. The higher of these abstractions is the level of entire machines, where we wish to consider machines as distinct objects with unique identities. The lower level of abstraction is the level of states and transitions. As might be expected, we will develop the higher level of abstraction (machines) in terms of the lower level of abstraction (states and transitions).

Following a discussion of machines (at both levels) and equivalence relations between machines, we define two notions of typing with respect to machines. We define a machine type to be a family of isomorphic machines, and we define an abstract type to be a family of behaviorally equivalent machines. These definitions will be the basis for our model of types. Finally, we will conclude the chapter by describing a variant of Petri Nets that we will use to illustrate the relationships between machines.

2.1 Algebra of States and Transitions: HCCS

It should be realized from the outset that the machine algebra that we will ultimately develop is simply syntactic sugar for a language of states and transitions: every expression in our machine algebra will have an exact equivalent in the language of states and transitions. In fact, we will conduct proofs of our theorems about our machine algebra by converting the machine algebra expressions into the corresponding language of states and transitions and conducting the proofs in that language. The advantage of the machine algebra is that one term in the machine algebra (one machine) will expand into a term *and a family of related equations* in the states and transitions language. The machine algebra formulation is thus more compact.

The states and transitions language used here is basically a hybrid of the operational models of Milner's CCS and Hoare's CSP. We shall call this hybrid language HCCS (for Hybrid-CCS). HCCS is CCS with the following modifications:

1 Instead of using CCS's flat set of actions, we use an abelian group of actions as employed in Milner's SCCS and ASCCS. This, in conjunction with the modified composition operator, will allow n-way synchronization between machines.

2 Milner's + operator is replaced by the similar □ operation used by Hoare. We will use $\oplus$ to represent this operation. This operator behaves the same as Milner's for deterministic choices, but differs in its treatment of non-deterministic choices. The use of the Hoare operator as opposed to the Milner operator makes Hoare's failures equivalence a congruence relation in HCCS.

3 We alter the definition of Milner's | operation to allow n-way synchronization between agents, where CCS only allows binary synchronization.

In the following sections, we summarize the important aspects of HCCS. The discussion in this chapter, while formal in places, is intended to provide an intuitive understanding of HCCS. Appendix A provides a formal summary of HCCS, including a semantic model based upon Milner's synchronization trees.

2.1.1 Actions

The state machines that we shall use in our model are simple extensions of classical finite state machines. Conceptually, each machine has a number of states, and a number of la-

beled transitions between states[1]. We shall call the labels on the transitions ***actions***, in keeping with Milner's terminology.

In classical finite state machines, each transition of a machine is labeled with a single symbol from a set of symbols, each one of which is primitive. In contrast to this flat structure of symbols, the set of actions has considerable structure to it. We begin with a set of ***atomic actions*** $\Lambda = \{a,b,c,...\}$, a set of ***atomic inverse actions*** $\overline{\Lambda} = \{\bar{a},\bar{b},\bar{c},...\}$, and a unique ***identity action*** 1. There is a 1:1 correspondence between atomic actions and inverse actions. The set of ***primitive actions*** $A = \Lambda \cup \overline{\Lambda} \cup \{1\} = \{..., \bar{c},\bar{b},\bar{a},1,a,b,c,...\}$. The complete set of actions is then constructed from the set of primitive actions, a ***composition operator*** $\times$ and an ***inverse operator*** $\bar{\ }$. The unique element 1 plays the role of an identity element with respect to the composition operator. We further note that every element has an inverse. The set of actions is then recursively defined by:

$A \subset Act$

$\forall a, b \in Act:$

1) $a \times b \in Act$

2) $\bar{a} \in Act$

The set of actions and its properties is more completely defined in Appendix A, section A.1 on page 134. We simply note here that Act is an Abelian group.

1. Unlike finite state machines, neither the number of states nor the number of actions is required to be finite.

We define $Atom(\mathsf{a})$ to be the set of primitive actions that a is comprised of. For example, if a and b are primitive actions, then $Atom(\mathsf{ab}) = \{\mathsf{a}, \mathsf{b}\}$. By extension, if A is a set of actions, we define $Atom(A) = \cup_{\mathsf{a} \in A} Atom(\mathsf{a})$. For $A \subseteq Act$, we define A^+ to be the set of actions generated by A and $\times$, and A^* be the subgroup generated by $A \cup \{1\}$, $\times$ and $\overline{\ }$.

As we shall soon see, actions are the means by which machines interact with each other. A machine can only perform an action a (take the transition labeled with the action a) when another machine with which it is interacting performs the inverse action $\overline{\mathsf{a}}$. If a machine has a transition labeled with the action ab, then it can only perform that action when other machines with which it is composed perform the actions $\overline{\mathsf{a}}$ and $\overline{\mathsf{b}}$. Note that this may either be a single machine performing the action $\overline{\mathsf{ab}}$, or two individual machines performing the actions $\overline{\mathsf{a}}$ and $\overline{\mathsf{b}}$.

2.1.2 Machines

At the level of states and actions, HCCS represents states with syntactic expressions that define the possible future behavior of the state. States come in parameterized and non-parameterized versions. Fully specified states are known as ***agents***. Parameterized states are known as ***agent expressions***. Agent expressions are allowed to contain variables, whereas agents are not. Thus a fully specified state machine is one in which all states (agents) are fully specified (contain no variables), whereas a parameterized state machine is one in which variables occur in one or more of the expressions defining its states (agents).

In developing our programming language semantics we shall need both fully specified state machines and machines whose specifications are parameterized. These parameterized machines we shall call ***generic state machines***, or generics for short. A generic can be con-

verted to a fully specified state machine by providing fully specified state machines as the actual values of the parameters.

In the following, we shall be using the term agent expression throughout in all definitions. Since an agent is simply an agent expression without variables, it should be understood that all of these definitions and related discussions pertain to agents as well. The discussions, up to but not including the discussion of equivalence, pertain uniformly to both fully specified state machines and generics. When we discuss equivalence, we shall have to treat generics differently than fully specified state machines.

A machine is specified in terms of its states and transitions. To specify a machine, we define a set of states that we shall call *agent expressions*, a set of labels that we shall call *actions*, a set of labeled transitions between states, and an initial state. The first three of these we shall refer to as a *labeled transition system*. More formally, a ***labeled transition system*** is a triple:

$$(\boldsymbol{E}, Act, \{\xrightarrow{a} : a \in Act\})$$

where $\boldsymbol{E}$ is a set of agent expressions, Act is a set of actions, and each $\xrightarrow{a}$ is a relation between agent expressions.

A ***machine*** is then a four-tuple consisting of a labeled transition system and an initial state P_0. Note that our machines, in contrast with classical finite state machines, do not necessarily have a finite number of states (agents) or a finite number of symbols (actions). A ***state machine*** is a machine in which no variables occur in its agent expressions. We shall use $\mathbf{M}_1$, $\mathbf{M}_2$, $\mathbf{M}_3$,... to represent state machines, and $\mathbf{M}$(x,y) to represent a generic machine having parameters x and y.

From time to time we shall need to compare sets of actions. In practice, we will usually take the set of atoms used in each set of actions and compare the sets of atoms instead. We shall call the set of atoms associated with a machine the ***sort*** of the machine.

We shall specify our labeled transition systems by giving families of equations that are similar to the production rules of a grammar. From this set of equations and a few inference rules we can derive the set of agents, set of actions and the set of relations that together define the labeled transition system.

2.1.3 Agent Expressions

The set of ***agent expressions*** $\boldsymbol{E}$ includes a number of ***agent constants*** $\boldsymbol{K} = \{A,B,C,...\}$ and a set of ***agent variables*** $\boldsymbol{X} = \{X,Y,Z,...\}$:

$$\boldsymbol{K} \cup \boldsymbol{X} \subseteq \boldsymbol{E}$$

Constants can be thought of as names that have been given to particular states (not all states are necessarily named with constants).

Given some initial members of the set of agent expressions $\boldsymbol{E}$, we now define the full set of agent expressions recursively as follows:

$$\forall\ E,F \in \boldsymbol{E},\ a \in Act:$$

Action:

1) $a.E \in \boldsymbol{E}$

Informally, $a.E$ means that the agent expression $a.E$ can perform the action a and then performs actions associated with the agent expression E.

Product:

2) $E \mid F \in \boldsymbol{E}$

$E \mid F$ means that the agent expressions E and F have been combined as machines operating in parallel with each other. Actions performed by the composition consist of either E or F performing an action (asynchronously, independent of the other agent expression) *or both* performing actions *simultaneously* (synchronous action). We shall provide a more formal description of the transition semantics later.

Summation:

3) $\sum_{i \in I} E_i \in \boldsymbol{E}$

where E_i is an independent set of agent expressions from $\boldsymbol{E}$, and I is an index set. $\sum_{i \in I} E_i$ means that the agent expression is a composition of alternative agent expressions, and will ultimately perform the observable actions specified by exactly one of the expressions in the set of E_i. There is one summation that we shall have occasion to use frequently, and this is the ***inaction machine* 0**:

$$\mathbf{0} \equiv \sum_{i \in \varnothing} E_i \qquad (1)$$

Because the set of expressions is empty, this machine never performs any actions.

In the special case of a summation involving two machines, we shall write the summation as $E \oplus F$.

It is important to note that the operation being defined here is *not* Milner's summation operator, but a generalization of Hoare's $\square$ operator to an arbitrary number of terms. As we shall see, the behavior of the Milner and Hoare operators is the same for observable actions, but different for the non-observable action 1[1].

Restriction:

4) $E \upharpoonright A \in \boldsymbol{E}$

$E \upharpoonright A$ means that the agent expression is the expression specified by E except that any actions not in $Atom(A)^*$ are hidden. If an action is hidden, then no agent expression outside of E may trigger that action. A related operation is to hide just the actions that are in $Atom(A)^*$. We shall designate this with the notation $E \setminus A$, and define this to be:

4a) $E \setminus A \equiv E \upharpoonright (Act - Atom(A)^*)$

Morphism:

5) $E[\phi] \in \boldsymbol{E}$

$E[\phi]$ means the agent expression specified by E after the actions have been mapped as specified by ϕ, where ϕ is a mapping from Act to Act such that $\phi(a) = \overline{\phi(a)}$ and $\phi(1) = 1$.

Constant Definitions:

1. As we shall see shortly, it is not possible for another machine to determine that a 1 action has occurred on another machine, hence the notion of non-observability.

Finally, we define constants by equating the constants with other agent expressions, as in the following example which defines the constant to be the agent expression E:

$$A \equiv E$$

Constants provide a means for defining recursive expressions. Consider the defining equations:

$$A \equiv a.B$$

$$B \equiv b.A$$

Together these equations define a machine that will endlessly perform the action sequence $a.b.a.b...$

2.1.4 Transition Rules

Now that we have a set of agent expressions, we give the rules defining the transitions between agent expressions.

Action:

$$\frac{}{a.E \xrightarrow{a} E} \qquad (2)$$

This rule says that the agent expression $a.E$ may make a transition to the agent expression E when the action a occurs. (We note that the occurrence of an action means that some other agent expression, with which this one has been composed in a product, has simultaneously made a transition with the action a.)

Summation:

Summation has two rules, the first covering observable transitions (transitions involving an action other than τ), and the other covering non-observable actions.

$$\frac{E_j \xrightarrow{a} E'_j}{\sum_{i\in I} E_i \xrightarrow{a} E'_j} \quad (j\in I, a\neq\tau)^{1} \tag{3}$$

This rule says that if agent expression E_j can be converted into an agent expression E'_j when the observable action a occurs, then a summation $\sum_{i\in I} E_i$ containing E_j can also be converted into the agent expression E'_j when the action a occurs. This gives summation the capability of representing a choice between the possible future states E_i. Once an action associated with one of these (E_j) has been made, the other choices are discarded and the summation behaves as E_j would have after making a transition on a.

$$\frac{E_j \xrightarrow{\tau} E'_j}{\sum_{i\in I} E_i \xrightarrow{\tau} \sum_{i\in I} E_i[E'_j/E_j]} \quad (j\in I)^{2} \tag{4}$$

This rule says that if an agent expression E_j can be converted into an agent expression E'_j when the unobservable action τ occurs, then a summation $\sum_{i\in I} E_i$ containing E_j can also be converted into the summation $\sum_{i\in I} E_i[E'_j/E_j]$ when the action τ occurs. Thus non-observable transitions in a summation do not cause the other alternatives to be discarded. Only

1. This is an obvious generalization of the ($COND_2$) inference rule of [Broo83] p. 168.
2. This inference rule is both less restrictive and more general than the ($COND_1$) inference rule of [Broo83] p. 168.

when one of the agent expressions in the summation makes an observable transition via Rule (3) are the other choices discarded.

Product:

There are three transition rules governing products. The first two cover the cases in which the machines in the product act asynchronously with respect to each other. The third rule covers the case in which the machines act synchronously.

$$\frac{E \xrightarrow{a} E'}{E \mid F \xrightarrow{a} E' \mid F} \tag{5}$$

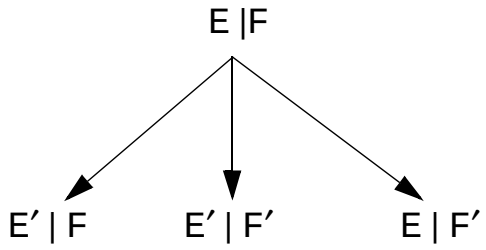
$$\frac{F \xrightarrow{b} F'}{E \mid F \xrightarrow{b} E \mid F'} \tag{6}$$

These rules say that if either machine can make a transition, then that machine may make the transition in the product. Here the other machine takes no action, and the machines operate asynchronously.

$$\frac{E \xrightarrow{a} E' \qquad F \xrightarrow{b} F'}{E \mid F \xrightarrow{ab} E' \mid F'} \tag{7}$$

This rule says that if the agent expression E can be converted into the agent expression E' when the action a occurs, and the agent expression F can be converted into the agent expression F' when the action b occurs, then the agent expression $E \mid F$ that is the product of these agent expressions can be converted into the agent expression $E' \mid F'$ when the action ab occurs (recall that ab is the product of the actions a and b).

One special case of this rule is of particular interest. If $a = \bar{b}$, then $ab = 1$ and the composition does not need to interact with any other machines in order to make the transition. The following diagram illustrates the transition possibilities of a product:



Restriction:

$$\frac{E \xrightarrow{a} E'}{E\restriction S \xrightarrow{a} E'\restriction S} \quad (a \in S) \tag{8}$$

where S is any subset of Act containing 1. This rule says that if the agent expression E can be converted into the agent expression E' when the action a occurs, then the agent expression $E\restriction S$ (which is the agent expression E with its actions restricted to the actions present in S) can be converted into the agent expression $E'\restriction S$ when the action a occurs provided that a is a member of S. The absence of any other rule for transition of restricted expressions implies is that if a is not a member of S then no transition is possible on a.

Morphism:

$$\frac{E \xrightarrow{a} E'}{E[\phi] \xrightarrow{\phi(a)} E'[\phi]} \quad (9)$$

This rule says that if the agent expression E can be converted into the agent expression E' when the action a occurs, then the agent expression $E[\phi]$ (the agent expression E with all of its actions mapped into new actions by the morphism ϕ) can be converted into the agent expression $E'[\phi]$ by the action $\phi(a)$.

Constant:

$$\frac{E \xrightarrow{a} E'}{C \xrightarrow{a} E'} \ (C \equiv E) \quad (10)$$

This rule says that if the constant C is defined to be the agent expression E, and E can make a transition to E' with the action a, then the constant C can also make a transition to E' with the action a.

2.1.5 Determining the Sort of a Machine

The sort of a machine is the set of primitive actions that occur in the labeled transition system of the machine. The ***sort*** of a state machine is the union of the sorts of the agents in its labeled transition system. Equivalently, the sort of the machine can be defined by *Atom(Act)*, the set of primitive actions that occur in the action set associated with the machine's labeled transition system.

The *Sort(*E*)* operation takes as its argument an agent E and returns the set of primitive actions that appear in the agent (the formal definitions of these operations and some resulting theorems appear in Appendix A section A.6 on page 147).

2.1.6 Constructing a Labeled Transition System from Equations

The transition rules define a relation between agents. We call the set of agents related to a given agent by an inference rule its ***successors***, and the transitive closure of the transition relation defines the set of ***descendants***. The set of agents in the labeled transition system is then the set containing the initial agent and all of its descendants. If A is the sort of the initial agent and its descendants, then A^* is a superset of the set of actions associated with the machine. The set of transitions is the set derivable from the initial agent and the transition rules.

2.1.7 Labeled Transition System Examples

We now give some simple examples of machines and their defining equations. Consider the following set of equations, which defines the state machine (in this case a finite state machine) shown in Figure 2.

$$A_1 \equiv a.A_2$$

$$A_2 \equiv b.A_1.$$

It should be pointed out that the set of equations defining a machine is not, in general, unique. This same machine could have been specified by the equation:

$$A_1 \equiv a.b.A_1$$

In this case, we have eliminated the constant A_2, but the machine structure is the same.

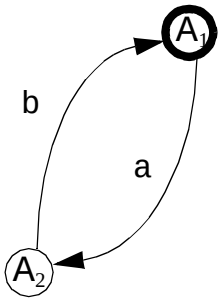


Figure 1 Simple State Machine

2.1.8 Interactions Between Machines

Interactions between state machines are simple: two machines interact when one makes a transition on an action while a second machine *simultaneously* makes a transition on the *inverse* of that action. Transitions are thus synchronous interactions between machines. *This is the only way in which machines may change state*.

As an example, consider the labeled transition systems of two machines as shown in Figure 2 and specified by the following set of equations:

Machine **A**:

$$A_1 \equiv a.A_2$$

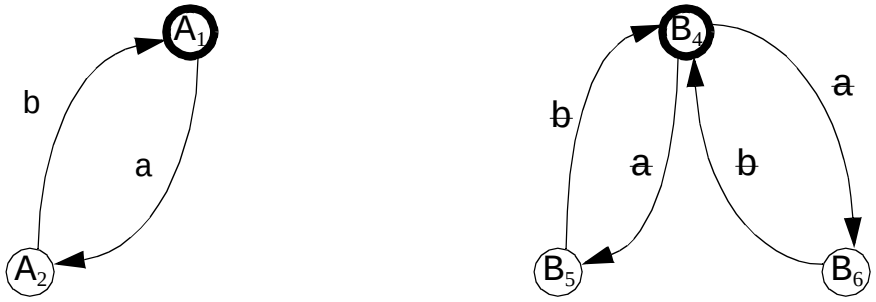
$$A_2 \equiv b.A_1$$

Machine **B**:

$$B_4 \equiv a.B_5 + a.B_6$$

$$B_5 \equiv \bar{b}.B_4$$

$$B_6 \equiv \bar{b}.B_4.$$



Machine **A** Machine **B**

Figure 2 Interacting State Machines

(Note that we have not given a complete definition of the machines, since we have not specified an initial state.) If machine **A** is in state A_1, and machine **B** is in state B_4, then machine **A** can make a transition to state A_2 with the a action at the same time that machine **B** makes a transition to state B_5 (or to state B_6) with the ~~a~~ action. Note that there may be non-determinism in machine behavior: machine **B** can transition to either state B_5 or state B_6 with the ~~a~~ action.

The joining of machines together so that interactions are possible we shall call the ***composition*** of machines, and we shall indicate composition with the symbol |, writing **A|B** for the composition of the machines **A** and **B** shown in Figure 2. Formally, this is defined to be the composition of the initial states of **A** and **B**, e.g. $A_1|B_4$ in the above example. The composition operator, like its agent counterpart, is both associative and commutative, but it is not idempotent.

2.2 Equivalence

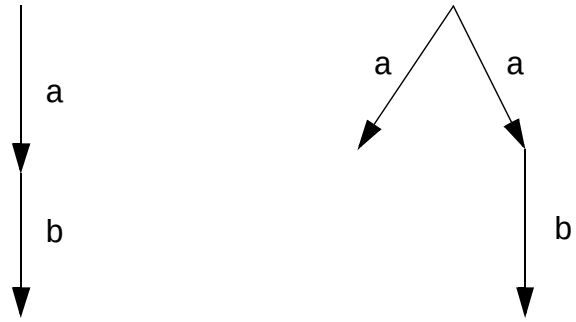
Just as important as the definition of a state machine is the notion of equivalence between two machines. We review by example the properties of several different equivalence relations stemming from automata theory, Milner's work on CCS, and Hoare's work on CSP. This comparison provides motivation for our use of Hoare's failures equivalence as our behavioral equivalence relation.

The notion of equivalence that we are interested in is one that is based upon our ability to distinguish between machines by observing differences in the way in which they respond to observable actions. In considering such equivalences, it must be noted that the only transition in a machine that *cannot* be influenced by the observer (the observer is, itself, simply another machine) and therefore cannot be observed, is the transition on the action 1, which Milner has called the *silent action*.

There are many possible notions of equivalence between machines, all differing mainly in their treatment of silent actions. While we shall be using only Hoare's failures equivalence [Hoare85] in our work, it is instructive to develop an informal understanding of this equivalence with respect to three other common notions of equivalence, namely *trace equivalence*[Miln89][McN82], *strong equivalence* [Miln80][Miln83][Miln89] and *observational equivalence* [Miln80][Miln83][Miln89].

The notion of equivalence arising from classical automata theory is often referred to as *trace equivalence*. Under trace equivalence, two machines are equivalent every sequence of actions that is possible for one machine is also possible for the other. Unfortunately, trace equivalence does not distinguish between the two machines shown in Figure 3, which we would certainly want to distinguish in a real design setting, since the non-determinism in machine **B** can lead to a state in which the action b would not be possible, while machine

A will *always* respond with the action b after the action a. We would not want to inadvertently substitute machine **B** (which has observable non-determinism) for machine **A** which is completely deterministic in its behavior.



Machine A **Machine B**

Figure 3 Trace Equivalent Machines

Milner's *strong equivalence* solves this problem by requiring that there be a very tight transition-by-transition correspondence between two machines in order to consider them equivalent. This notion of equivalence turns out to be too strong for our purposes. Although strong equivalence distinguishes between the machines of Figure 3 (as we wish), it also distinguishes between the machines of Figure 4, which in a typical design context we would want to consider equivalent since their observable actions are identical.

Milner's weak or *observational equivalence* is a step closer to what we are looking for, since it ignores 1 actions that do not affect branching behavior as in Figure 4. Unfortunately, it still requires a structural similarity between machines that is stronger than what we desire. Figure 5 shows two machines that we would like to consider equivalent, but which observational equivalence distinguishes between[1]. Note that both machines have exactly the same amount of non-determinism, but that the branches that decide the restrictions on

Figure 4 Machines Distinguished by Strong Equivalence

a and b simply occur in a different order. There is no way to distinguish between these machines through experimentation, and therefore we would like to consider them equivalent.

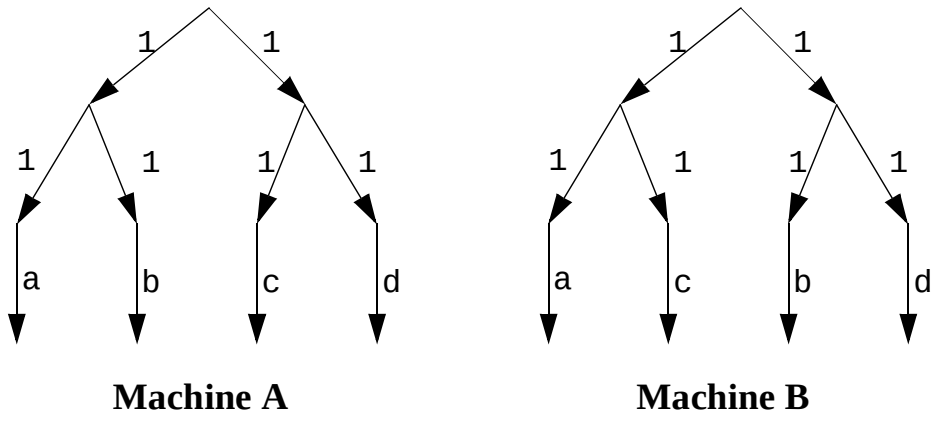


Figure 5 Machines Distinguished by Observational Equivalence

1. Our model was originally formulated using observational equivalence, but Theorem 4 could not be proved in this system. It was this example that Milner used to refute Theorem 4 in the original formulation and it was his suggestion that Hoare's failures equivalence might be more appropriate for our model.

Finally, we arrive at Hoare's notion of failures equivalence, which we shall call behavioral equivalence. Behavioral equivalence distinguishes between the machines of Figure 3, while considering the machines of Figure 4 equivalent and the machines of Figure 5 equivalent. We will shortly provide a definition for behavioral equivalence.

There is a well defined relationship between these four equivalences. Strong equivalence implies observational equivalence, which implies behavioral equivalence, which implies trace equivalence.

2.2.1 Behavioral Equivalence

Behavioral equivalence (which is Hoare's notion of failures equivalence) considers two aspects of the behavior of a machine:

1 The traces of a machine (the sequences of actions that it can generate);

2 The failures of a machine, which are the actions that the machine *may not* respond to after a given trace.

It is important to recognize that these conditions characterize what a machine *may* do, not necessarily what it *will* do. The traces of a machine indicate action sequences that the machine may generate. The failures of a machine indicate the actions that may be rejected after the machine has responded to a sequence of actions.

Formally, Hoare characterizes a machine in terms of its *observable actions*, its *traces* (observable sequences of actions) and its *failures* (the actions that may not be responded to after a given sequence of observable actions).

The ***initials*** of a machine are the first observable actions of the machine. The ***traces*** of the machine are the sequences of observable actions that occur starting from the initial state of

the machine. A machine can ***refuse*** a set of events X if it can make a silent transition to a state none of whose initials is a member of X[1]. The ***failures*** of a machine are the pairs (s, X) such that the machine has a state reachable through the observable sequence s that can refuse X. The following definitions[2] make this precise

$$\mathit{Initials}(S) = \{a{:}\in Act \mid \exists S' \text{s.t. } S \overset{\langle a \rangle}{\Rightarrow} S'\}$$

$$\mathit{Traces}(S) = \{s \mid \exists S' \text{s.t. } S \overset{s}{\Rightarrow} S'\}$$

$$\mathit{Refusals}(S) = \{X \mid \exists S' \text{s.t. } S \overset{\langle\rangle}{\Rightarrow} S' \text{ and } X \cap \mathit{Initials}(S') = \varnothing\}$$

$$\mathit{Failures}(S) = \{(s,X) \mid \exists S' \text{s.t. } S \overset{s}{\Rightarrow} S' \text{ and } X \cap \mathit{Initials}(S') = \varnothing\}$$

We have the following result for behavioral equivalence:

$$P = 1.P$$

2.2.2 Equivalence of Generic Machines and Agent Expressions

As we pointed out earlier, we must make a distinction between agents and agent expressions (state machines and generic machines) when defining equivalence. We cannot directly compare the sequences of actions and failures of agent expressions since we will not know what all of the sequences will be until values have been provided for all of the variables. Consequently, we define two agent expressions (generic machines) to be equivalent

1. Note that every state can make a silent transition to itself
2. [Broo83] p. 96

if they have the same set of variables and, for all possible values of the variables (where the values are themselves agents) the agents that result from the substitution are equivalent.

Consider the following three generic machines, in which X is a variable:

$$G_1 \equiv a.b.0 \mid X$$

$$G_2 \equiv (a.b.0 \oplus a.b.0) \mid X$$

$$G_3 \equiv (a.b.0 \oplus a.0) \mid X$$

Machines G_1 and G_2 are equivalent for all values of X, since $a.b.0$ is behaviorally equivalent to $(a.b.0 \oplus a.b.0)$ and each value of X, by definition, is equivalent to itself. Machine G_3, on the other hand, is never behaviorally equivalent to either of the other machines, since $(a.b.0 \oplus a.0)$ is not behaviorally equivalent to either $a.b.0$ or $(a.b.0 \oplus a.b.0)$.

2.3 Additional Machine Properties and Constraints

The calculus of machines that we have developed thus far is capable of defining a very broad class of machines. However, our model of programming languages is not based upon the use of arbitrary machines, but rather upon machines with very specific properties. In this section we will define those additional properties and provide some motivation for their usefulness.

Since one of the major goals of our work is to formulate a semantic model that is modular in the sense that each machine can be understood independently of other machines[1], we

1. This is not to say that the interaction between machines is not a factor in understanding the machines, but rather to say that the definition of one machine should be independent of the definition of others.

must ensure that our machine specifications (formulated in HCCS) are mutually independent. We therefore impose a constraint that the constants used in the defining equations of each state type be unique to that state type. This will ensure that each state type's defining equations are distinct from those of every other state type.

Up to this point, we have not ascribed any meaning to any of the actions associated with our machines. Without ascribing meanings, these models are of little benefit, so we begin by categorizing the actions and ascribing meaning to the categories.

2.3.1 Input and Output Actions

Back when we first defined actions, we constructed the set of primitive actions from a set of atomic actions and a set of atomic inverse actions. We will use atomic actions to represent the basic operations that a machine is capable of performing. For example, if we had a machine representing a variable, then the actions that set values and read values on the variable would consist entirely of actions composed of atomic actions. We shall call these actions ***input actions***, because they represent an input to the machine to ask it to do something. Note that the term input refers to the direction of the request, not the direction of information flow: the action reading the value of a variable goes *to* the variable, even though the net result is the transfer of a value *out of* the variable. We note that the set of input actions is simply Λ^+, the set of all actions constructed from atomic actions.

The inverse of an input action is an ***output action***. An output action is an action constructed entirely of atomic inverse actions, and thus output actions are all members of the set $\overline{\Lambda}^+$. An output action represents a requests by a machine for another machine to take some action. For example, a machine representing an assignment operator would make a request of one variable to read its value, and another variable to store the value.

There are members of the set of actions that are neither input actions or output actions: they contain both atomic actions and atomic inverse actions. We shall call such actions ***mixed actions***.

2.3.2 Atomic Actions Appear on Exactly One Machine

Atomic actions represent a request to a machine to perform some operation. Since we wish such requests to be unambiguous, we shall require that ***an atomic action may appear as a transition label on exactly one machine*** (although it may label any number of arcs on that machine). In contrast, the inverse of that action may appear on any number of machines. We will refer to the one machine on which an action appears as the ***defining machine*** of that action. We define the *Def()* operation that takes a machine as an argument and returns the set of actions defined on that machine.

2.3.3 Activate and Idle Actions

One of the important issues in our model is the question of when machines are created and destroyed. We shall deal with this issue in a formal manner by considering that *all* machines exist *all* the time. Instead of creating and destroying the machines, we will add an additional state to each machine, its `IDLE` state, and two additional actions, the ***activate action*** α and the ***idle action*** ι, to each machine. While in the `IDLE` state a machine never responds to any action except the activate action. An interaction with the activate action is equivalent to creating the machine, and causes the machine to transition from the idle state to its initial working state. An interaction with the idle action is equivalent to destroying the machine, and causes the machine to transition back to the idle state. We note that the activate and idle actions are both atomic actions, and are therefore unique to the machine that they appear on.

For the purposes of defining machines, we subdivide the set of actions Act into three disjoint subsets: Act_α, Act_ι, and Act_ω, where Act_α is the set of activation actions, Act_ι is the set of idle actions, and Act_ω is the set of all other actions which we shall refer to as working actions. We thus have,

$$Act = Act_\alpha \cup Act_\iota \cup Act_\omega \cup \{1\}$$

2.4 Orthogonality

It is often important to determine if two machines can *not* directly interact with each other, since if two machines cannot interact (directly or indirectly), then a number of equivalent restructurings of a composition in which the machines occur may be possible.

This condition of relative independence is so important that we give it a name: ***orthogonality***. If no actions appear on one machine whose inverses appear on the other (we allow both to have the silent action 1), we say that they are ***orthogonal***. We use the symbol $\perp$ to denote orthogonality, and write

$$\mathbf{M}_i \perp \mathbf{M}_j$$

to indicate that $\mathbf{M}_i$ and $\mathbf{M}_j$ are orthogonal. Formally the orthogonality relation is given by:

$$\mathbf{M}_i \perp \mathbf{M}_j \Leftrightarrow Sort(\mathbf{M}_i) \cap \overline{Sort}(\mathbf{M}_j) = \overline{Sort}(\mathbf{M}_i) \cap Sort(\mathbf{M}_j) = Sort(\mathbf{M}_i) \cap Sort(\mathbf{M}_j) = \{1\} \quad (11)$$

For convenience,we shall also define orthogonality between subgroups of actions, and between the actions of a machine and an arbitrary subgroup of actions:

$$S \perp T \Leftrightarrow Atom(S) \cap \overline{Atom}(T) = \overline{Atom}(S) \cap Atom(T) = Atom(S) \cap Atom(T) = \{1\}$$

$$\mathbf{M}_i \perp S \Leftrightarrow Sort(\mathbf{M}_i) \cap \overline{Atom}(S) = \overline{Sort}(\mathbf{M}_i) \cap Atom(S) = Sort(\mathbf{M}_i) \cap Atom(S) = \{1\}$$

where $S, T \subseteq Act$.

We have the following result for orthogonality:

$$\mathbf{M}_j \perp \mathbf{M}_a, \mathbf{M}_j \perp \mathbf{M}_b \text{ implies } \mathbf{M}_j \perp (\mathbf{M}_a \mid \mathbf{M}_b)$$

2.5 Types

In this section, we will explore the notion of a ***type*** as an *equivalence class* of machines. There are two different kinds of equivalence class that we shall find useful in our work. The first of these, which we shall call a ***state type***, is an equivalence class in which all machines belonging to the class are *isomorphic* under a mapping operation on actions. The second kind of equivalence class we shall call an ***abstract type***, in which the machines belonging to the type are *behaviorally equivalent* under a mapping operation on actions when interaction with the machines is restricted to the actions associated with the type, but are not necessarily isomorphic. As an example of the difference between state types and abstract types, consider the following three machines:

$$A_1 \equiv a.\mathbf{0}$$

$$A_2 \equiv a.\mathbf{0} \oplus a.\mathbf{0}$$

$$A_3 \equiv a.\mathbf{0} \oplus b.\mathbf{0}$$

There is no mapping between any pair of these machines that makes them isomorphic, but machines A_1 and A_2 are clearly behaviorally equivalent. Machine A_3 can be made behaviorally equivalent to A_1 and A_2 by restricting its actions to the set $\{a\}$. Thus all three ma-

chines could be members of an abstract type defined as a set of machines behaviorally equivalent to A_1 and restricted to the actions of {a}.

Abstract types give us a way to extend the behavior of machines while still maintaining a well-defined relationship with other classes of machines. Abstract types can be used to define hierarchies of types, in which the "parent" class defines common behavior (its abstract type) for all of its children. Note that a machine may belong to more than one abstract type. For example, if we define the machine A_4:

$$A_4 \equiv b.\mathbf{0}$$

and use it as the basis for defining a second abstract type (this time restricted to the actions of {b}), then machine A_3 can be a member of both abstract types. This ability to be a member of more than one abstract type provides a nice model for multiple inheritance in object oriented languages. Abstract types also provide a model for the architecture/implementation concepts found in VHDL.

2.5.1 State Types

A state type is a set of state machines that are isomorphic under a mapping operation on actions. Although the labeling of each machine in the set must be different (at a minimum, the activate and idle actions must be different), the labeling varies from machine to machine within the set in a very structured and regular way. For any two machines $\mathbf{M}_i$ and $\mathbf{M}_j$, there is a mapping from the actions of $\mathbf{M}_i$ to the actions of $\mathbf{M}_j$ such that if the mapping is used to relabel $\mathbf{M}_i$, the resulting machine is identical in all respects to $\mathbf{M}_j$. All machines in the set are thus isomorphic. Figure 6 shows a set of four isomorphic machines.

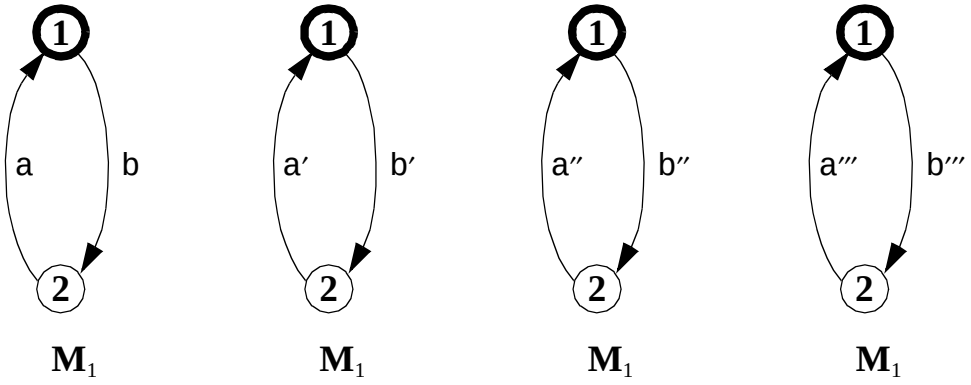


Figure 6 Four Isomorphic Machines

More formally, a ***state type*** $\boldsymbol{S}$ is defined to be a triple (M,L,Φ), where $M = \{\mathbf{M}_1, \mathbf{M}_2, \mathbf{M}_3,...\}$ is an indexed set of state machines (we will let $I = \{i \mid \mathbf{M}_i \in M\}$ be the set of indices of the machines in the state type), $L = \cup_{i\in I} Act(\mathbf{M}_i)$ is the set of actions associated with the state type,and $\Phi = \{\phi_{j\to k}: Act(\mathbf{M}_i) \to Act(\mathbf{M}_i) \mid j,k \in I\}$ is a set of morphisms mapping the actions of each machine to the actions of every other machine in the type such that for all $\mathbf{M}_j, \mathbf{M}_k \in M$:

$$\mathbf{M}_j[\phi_{j\to k}] \equiv \mathbf{M}_k$$

where $\mathbf{M}_j[\phi_{j\to k}]$ indicates the relabeling of $\mathbf{M}_j$ to use the actions of $\mathbf{M}_k$, and $\equiv$ signifies identity. As a notational convenience, we shall often write $\mathbf{M}_j[\phi_k]$ when we mean $\mathbf{M}_j[\phi_{j\to k}]$. As a further notational convenience, we shall frequently write $\mathbf{M} \in \boldsymbol{S}$, where $\boldsymbol{S}$ is a state type, when we mean that $\mathbf{M} \in M_{\boldsymbol{S}}$, where $M_{\boldsymbol{S}}$ is the set of machines associated with that state type.

While this may be a satisfactory definition of a state type from a mathematical perspective, we have not yet provided a constructive means of *specifying* the machines that belong to the state type. We thus give an alternate and equivalent definition of state type as a pair $(\mathbf{M}_0,\phi)$ where $\mathbf{M}_0$ is a fully specified "prototype" state machine and $\phi = \{Act(\mathbf{M}_0) \to Act(\mathbf{M}_n)\}$

is a set of functions mapping the actions of $\mathbf{M}_0$ to the actions of $\mathbf{M}_1$, $\mathbf{M}_2$, $\mathbf{M}_3$,..., which are the other machines of the type such that for all $\mathbf{M}_k \in \boldsymbol{S}$:

$$\mathbf{M}_0[\phi_{0\to k}] \equiv \mathbf{M}_k$$

where $\mathbf{M}_0[\phi_{0\to k}]$ is a relabeling of $\mathbf{M}_0$ to use the actions of $\mathbf{M}_k$ and $\equiv$ signifies identity.

2.5.2 Abstract Types

In the previous section, we defined a state type to be a set of machines that were isomorphic under a relabeling operation. We now define a less restrictive notion of type that we shall call an ***abstract type***. Instead of requiring that the machines be *isomorphic* under the relabeling operation, we merely require that the machines of an abstract type be *behaviorally equivalent* under the relabeling operation when interaction with the machines is restricted to the actions associated with the type.

Formally, an abstract type $\boldsymbol{A}$ is defined to be a triple (M,L,Φ), where $M = \{\mathbf{M}_1, \mathbf{M}_2, \mathbf{M}_3,...\}$ is an indexed set of state machines (we will let $I = \{i \mid \mathbf{M}_i \in \boldsymbol{M}\}$ be the set of indices of the machines in the state type), L is the set of actions associated with the abstract type,and $\Phi=\{\phi_{j\to k}:(Act(\mathbf{M}_j) \cap L) \to (Act(\mathbf{M}_k) \cap L) \mid j,k\in I\}$ is a set of functions mapping those actions of each machine that belong to the abstract type to those actions of the other machines that also belong to the type such that for all $\mathbf{M}_j, \mathbf{M}_k \in \boldsymbol{S}$:

$$\mathbf{M}_j[\phi_{j\to k}] \upharpoonright L = \mathbf{M}_k \upharpoonright L$$

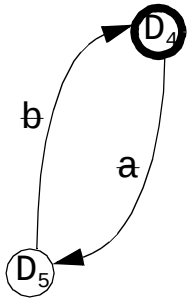
where $=$ signifies behavioral equivalence.

Lemma 1 Every State Type is also an Abstract Type

We now show that every state type is also an abstract type, with M_A=M_S, L_A = L_S and Φ_A=Φ_S.

Proof: Let $\boldsymbol{S}$=(M,L,Φ) be a state type, and M_j and $\mathsf{M}_k \in \boldsymbol{S}$. Then $\mathsf{M}_j[\phi_{j\to k}]\restriction L \equiv \mathsf{M}_k \equiv \mathsf{M}_k\restriction L$ (where ≡ signifies identity), and since identity implies failures equivalence, then $\mathsf{M}_j[\phi_{j\to k}]\restriction L = \mathsf{M}_k\restriction L$ under behavioral equivalence, hence (M,L,Φ) is also an abstract type.

Based upon this definition, it should be obvious that more than one state type can belong to the same abstract type. For example, machine **D** in Figure 7 is behaviorally equivalent to machine **B** in Figure 2, and therefore both could be members of the same abstract type.



***Machine* D**

Figure 7 Behaviorally Equivalent Machine for Machine B

2.5.3 Extensionality and Types

It is reasonable to ask how the membership of machines in state types and abstract types is determined. One strategy is to explicitly specify the membership of each machine in the type. With this approach, the mere *existence* of a mapping between two machines that would make them equivalent (either isomorphic or behaviorally equivalent) under a rela-

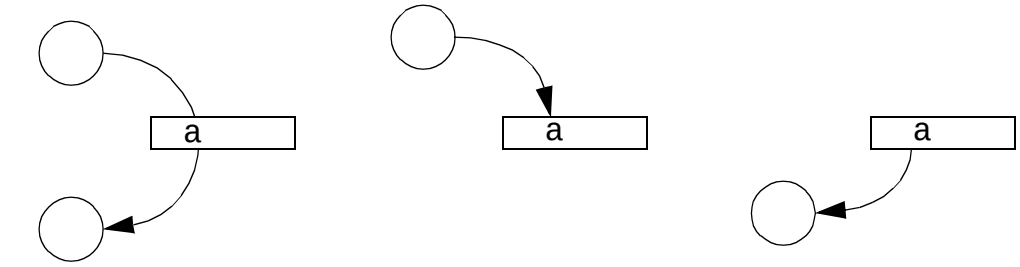
beling operation does *not* mean that the machines are of the same (state or abstract) type. We believe that this approach is consistent with the use of types in production programming languages.

Another strategy is to consider machines to be of the same type if a mapping exists that makes them equivalent. This is analogous to the principle of extensionality discussed frequently in the denotational semantics literature. We believe that this approach is inconsistent with the use of types in production programming languages. Furthermore, we conjecture that this notion is incompatible with the notion of function overloading in the sense that it will make overload resolution undecidable in many cases.

2.6 Petri Net Extensions

For ease in illustrating the interactions between machines, we shall develop a graphical notation based on petri nets that we shall call *reduced Petri nets*. Reduced Petri nets are colored Petri nets with two extensions added. The first of these is due to Eaker [Eaker91a][Eaker91b]. In Eaker's notational extensions (see Figure 8), the identity of each token is preserved as it passes through a transition. This is indicated by an arc passing *through* the transition (Figure 8a). Similarly, the notation indicates the destruction of tokens (Figure 8b) and the creation of tokens (Figure 8c). We further extend Eaker's notation by labeling each intersection of an arc and a transition with an action. Note that since the only possible interaction between machines is between a machine with an arc labeled with an action a and another machine labeled with its inverse $\bar{a}$, each transition in our diagrams will be labeled with both an action and its inverse.

Figure 9 uses our reduced Petri net notation to illustrate the interaction of machines A and B from Figure 2. In this illustration there is a separate transition in the diagram for each



a) Token preserved through transition ***b) Token destroyed at transition*** ***c) Token created at transition***

Figure 8 Eaker's Extensions to Petri Nets

possible interaction between the machines **A** and **B**. The presence of a separate transition for each interaction leads to a combinatorial explosion that significantly limits the usefulness of the extended Petri net diagram as a documentation technique. This naturally leads us to consider possible ways of simplifying the diagram.

2.6.1 Reduced Petri Nets

The simplification we shall consider here involves reducing a collection of nodes in the Petri net to a single node containing a typed token. We can take the set of nodes that a token can pass through in a Petri net along with the set of labeled arcs connecting the nodes to be the definition of a machine **M**. We can then collapse these nodes into a single node on the Petri net containing the machine **M**. Similarly, we can collapse all of the transitions into a single transition that is now labeled with the actions of the machine *Act(***M***)*. To be equivalent to the original Petri net, the new node must contain a single instance of machine **M**.

Now the collapsing of the transitions will require a similar collapsing of the nodes on the other side into its machine definition. We call a network that has been reduced in this manner a ***reduced Petri net***. While reductions of this type are not always possible in arbitrary Petri nets, we shall see that the nets arising from our semantic model will frequently be re-

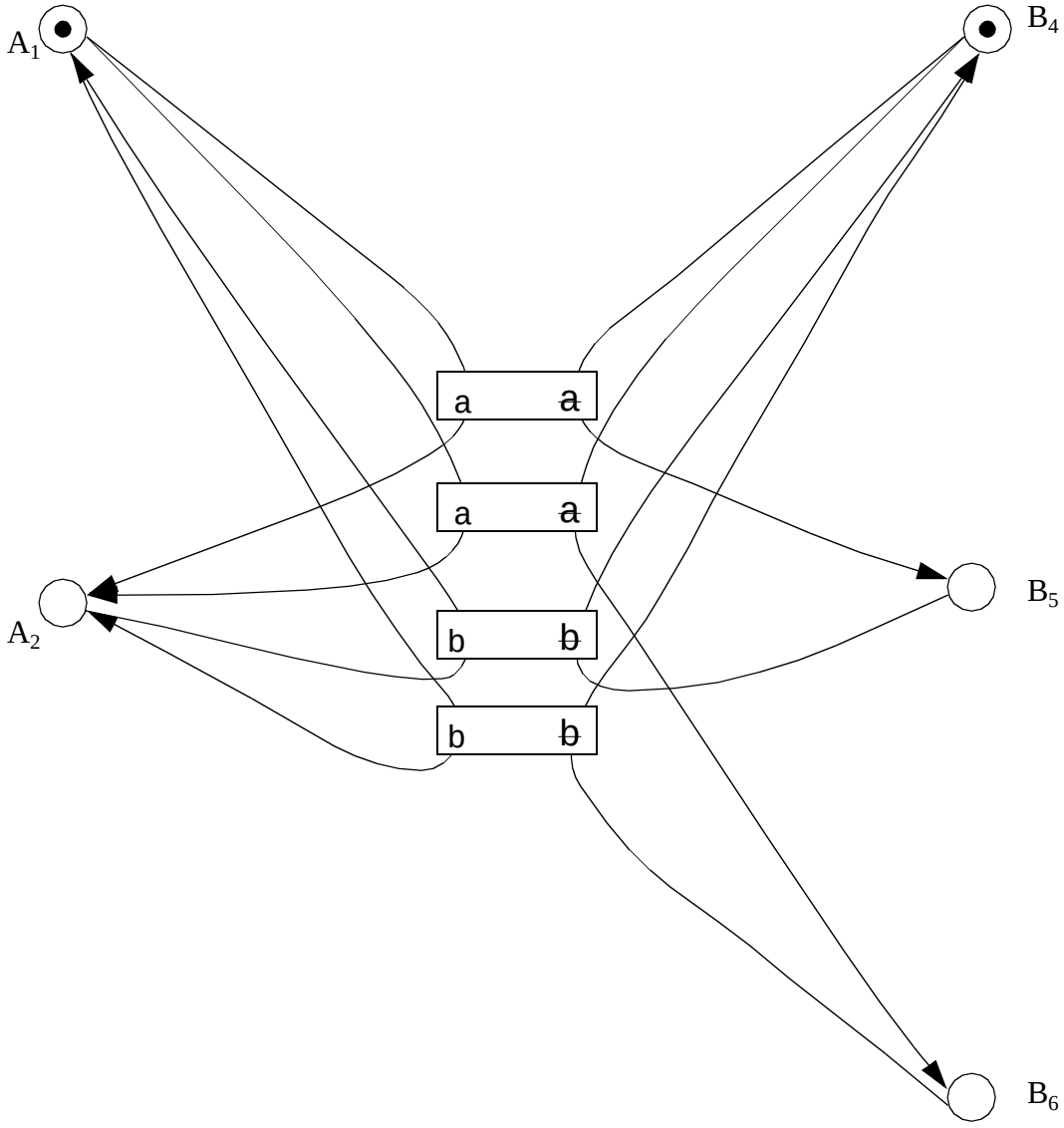


Figure 9 Extended Petri Net Showing the Composition A|B

ducible. In fact, we shall see that this kind of reducibility is highly desirable: the ability to reduce in this manner is the ability to present a description of a piece of software at a higher level of abstraction.

For example, the composition **A|B** of Figure 9 can be represented by the reduced Petri Net of Figure 10, with $L_A = \{a, b\}$, and $L_B = \overline{L}_A = \{\overline{a}, \overline{b}\}$.

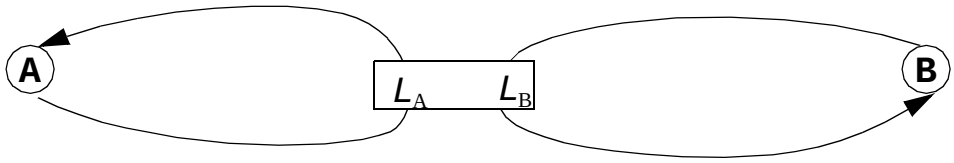


Figure 10 Reduced Petri Net Showing the Composition A|B

3.0 Classes of Machines

In the previous chapter we hinted that the building blocks for our programming language semantics would not be based upon arbitrary machines, but rather a smaller class of machines having specific properties. Two of these properties that are shared by all of the machines that we will use have already been discussed: all machines may operate asynchronously with respect to each other; and atomic actions (as opposed to their inverses) appear on exactly one machine.

In defining our semantics we will require two categories of machines: ***value machines***, which are used to model the storage of values; and ***interaction machines***, which are used to model interactions between other machines.

3.1 Value Machines

One of the fundamental needs of a programming language is to have a means of representing values. In our model of programming languages, each variable is modeled as an instance of a special kind of state machine that we shall call a *value machine*. The states of this machine represent the values that the variable can assume. A data type in a programming language then corresponds to a state type that defines the *collection* of machines capable of representing those values. In a simplistic model of programming languages, declaring a variable to be of a particular type can be interpreted as designating the particular machine from the state type that will represent this particular variable. As we shall see in the next chapter, a more complex interpretation will be required when type structure is introduced.

To clarify the intended effect of actions on variable machines, we subdivide the working actions of value machines into two subsets: one set of actions that *set* values, and one set of

actions to *read* the values. We shall call the actions that set values the *input actions*, and designate these actions with the notation

$$\mathtt{i}n$$

where i indicates that this is an input action (one that sets a value), and *n* indicates that this is the *n*th input action associated with this machine (i.e. is associated with the *n*th value).

The *output actions* of the machine represent actions that *read* the value of the variable. In a manner similar to the input actions, we designate the output actions with the notation

$$\mathtt{o}n$$

As an example, consider the machine defined by the following equations and shown in Figure 11:

$$V_1 \equiv \mathtt{o1}.V_1 \oplus \mathtt{i1}.V_1 \oplus \mathtt{i2}.V2 \oplus \mathtt{i3}.V_3$$

$$V_2 \equiv \mathtt{o2}.V_2 \oplus \mathtt{i1}.V_1 \oplus \mathtt{i2}.V_2 \oplus \mathtt{i3}.V_3$$

$$V_3 \equiv \mathtt{o3}.V_3 \oplus \mathtt{i1}.V_1 \oplus \mathtt{i2}.V_2 \oplus \mathtt{i3}.V_3$$

This machine is suitable for representing a variable that can assume one of three possible values. Each of the agents V_1, V_2 and V_3 represents a value. From any given state the machine can be made to assume any other state (may represent any value) given the proper input action (value to assume). Note that in any particular state, the only output action that may occur (value that may be read) is the output action corresponding to the current state: i.e. the value read is the value that the machine is currently representing.

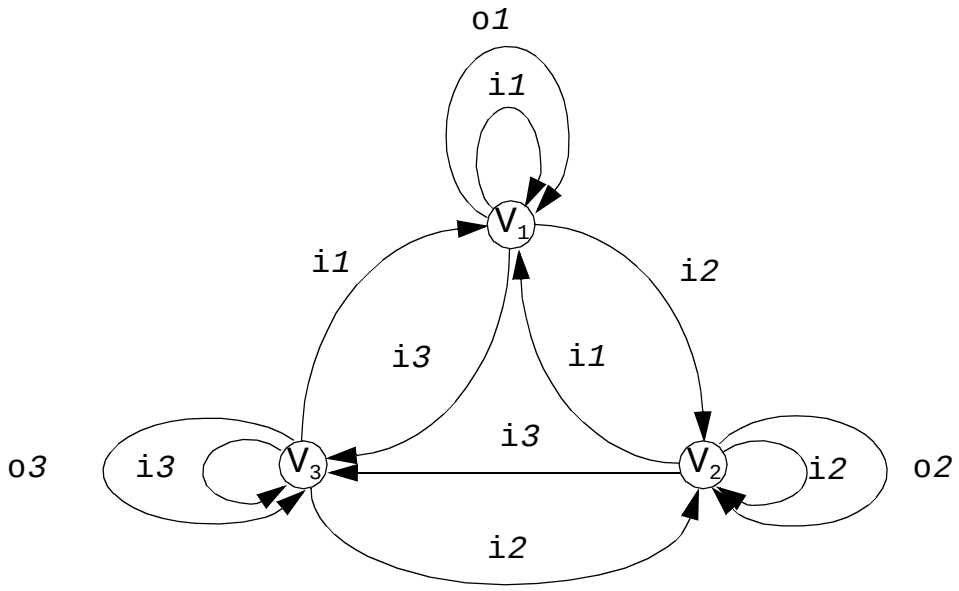


Figure 11 Incomplete Value Machine for 3-valued Variable

3.1.1 Value Machines are Mutually Orthogonal

Intuitively, we would expect that, in the absence of any other machines, operations performed on one value machine would not affect the state of any other value machine. We note that value machines, as we have defined them, are labeled entirely with primitive actions and do not have any inverse actions. We further note that we have required that all primitive actions except for 1 appear on exactly one machine. Thus we have:

Lemma 2 All value machines are mutually orthogonal.

3.1.2 Value Machine Activation and Idling

In our initial description of the value machine, we neglected the activate and idle actions, and the idle state. The complete value machine, with initial state V_0, is given by the following equations, and is illustrated in Figure 12.

$V_0 \equiv \alpha.V_1 \oplus \alpha.V_2 \oplus \alpha.V_3$

$V_1 \equiv o1.V_1 \oplus i1.V_1 \oplus i2.V2 \oplus i3.V_3 \oplus \iota.V_0$

$V_2 \equiv o2.V_2 \oplus i1.V_1 \oplus i2.V_2 \oplus i3.V_3 \oplus \iota.V_0$

$V_3 \equiv o3.V_3 \oplus i1.V_1 \oplus i2.V_2 \oplus i3.V_3 \oplus \iota.V_0$

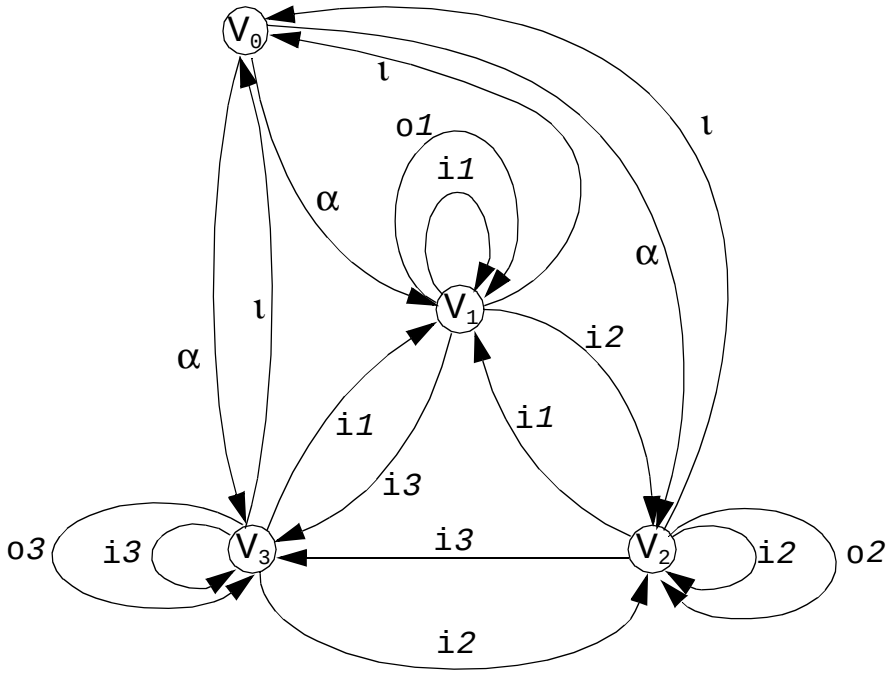


Figure 12 Complete Value Machine for 3-valued Variable

Note that when the activate action α occurs, the initial state of the variable is non-deterministic. One could easily define an alternate semantics for variables in which the machine deterministically transitioned to a specified state upon activation. Also note that the idle action ι places the machine back in its original state, and that there is no carryover of state

information once the idle action has occurred. Thus the activate action is equivalent to the creation of a new machine, and the idle action is equivalent to destroying a machine.

While the structure of these machines is simple, it is clear that using diagrams such as Figure 12 for each variable will lead to complexities in diagrams that show interactions between machines similar to those that are illustrated in Figure 9. Consequently, we shall use our reduced Petri net notation, collapsing all of the value states into a single node in the Petri net, and labeling this node with the *abstract* type of the variable. From this node we shall generally show two types of transitions, one labeled with the `in` actions, and the other labeled with the `out` actions, giving us the reduced Petri net of Figure 13.For clarity in these diagrams we will show the state prior to activation and the state after idling as separate states. This is without loss of generality, since there is no state information carried through from idling to activation. In addition to the simplification shown here, we will frequently use the Eaker notation to show the creation and destruction of tokens.

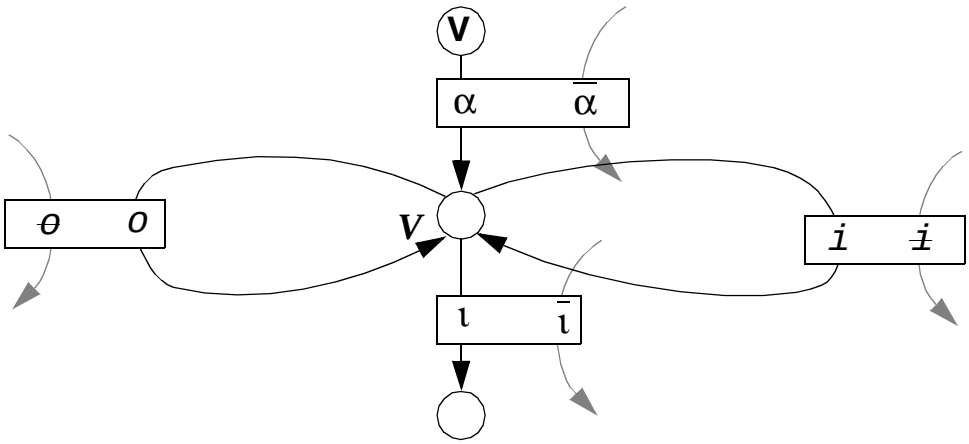


Figure 13 Reduced Petri Net Fragment Showing a Variable of Type *V*

3.1.3 Constants

A constant is simply a special case of a value machine with a single activate transition and all of the input omitted. An example of a constant for the first value of our three-valued data type is shown in Figure 14. Figure 15 shows a typed Petri net for a constant of type ***V***.

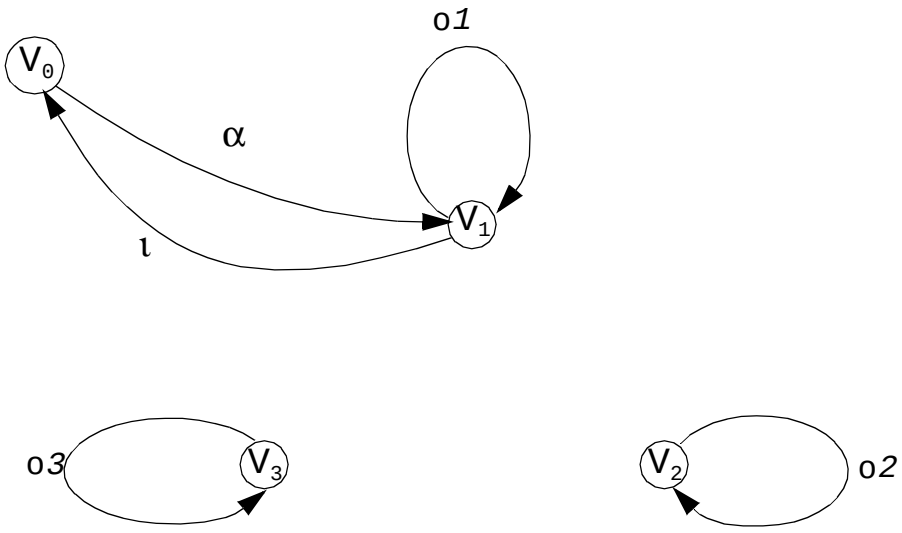


Figure 14 Value Machine for Constant "1"

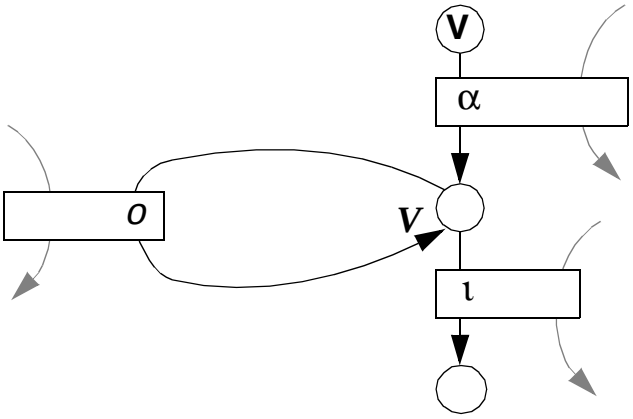


Figure 15 Reduced Petri Net Fragment Showing a Constant of Type *V*

The formal specification of this constant is given by:

$$C_0 \equiv \alpha.C_1$$

$$C_1 \equiv o1.C_1 \oplus \iota.C_0$$

$$C_2 \equiv o2.C_2 \oplus \iota.C_0$$

$$C_3 \equiv o3.C_3 \oplus \iota.C_0$$

Note that the definitions of C_2 and C_3 are superfluous, since neither one is reachable from the initial state. We show them primarily to clarify the relationship between value machines and constant machines.

3.2 Interaction Machines

Value machines are machines that store values. Since they cannot interact with each other (we have already shown them to be mutually orthogonal) we must introduce another class of machine which we shall call ***interaction machines*** expressly for this purpose. In order to fulfill their role, interaction machines must be labeled with the inverse actions. The only *new* actions (non-inverse actions) that are introduced on an interaction machine are their activate and idle actions. Thus the only orthogonality that is guaranteed between interaction machines, and between interaction machines and value machines is that the activate and idle actions are different on each machine, this stemming from our requirement that atomic actions only appear on only one machine.

As we shall see, a significant property of interaction machines is that they *never* encode values in their internal state: the machines simply serve as catalysts for the transfer of values from one value machine to another.

We shall adopt a naming convention for interaction machines that is intended to make our machine algebra expressions more readable. In general, we will use the notation

$$\mathbf{F}_x^{j\to k}$$

to designate an interaction machine, where $\mathbf{F}$ indicates kind of interaction machine this is (for example, we will use $\mathbf{=}$ to designate an assignment machine), the subscript x uniquely designates this machine (we may well have several assignment machines around), and $^{j\to k}$ is an optional *informal* indication of which machines this one interacts with.

We will use interaction machines to model many operations on basic data types. For example, the assignment statement of a basic data type is modeled as an interaction machine that reads a value from a value machine and sets the value of a (usually different) value machine. A state diagram for a typical assignment machine $=^{j\to k}$ for reading three-valued variable $\mathbf{M}_j$ and setting three-valued variable $\mathbf{M}_k$ is shown in Figure 16. The subscripts indicate the defining machines for each action.:

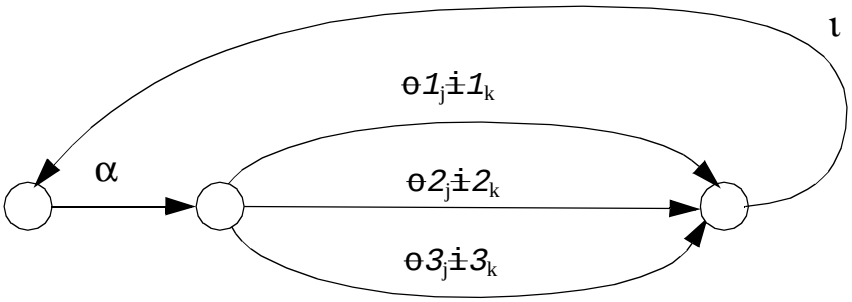


Figure 16 Assignment Machine $=^{j\to k}$

This machine is defined by the following expression:

$$=^{j\to k} \equiv E_0 \equiv \alpha.(ө1_j і1_k.\iota.E_0 + ө2_j і2_k.\iota.E_0 + ө3_j і3_k.\iota.E_0)$$

where the activate and idle actions are unique to this machine. As before, we will more often use a typed Petri net representation for the assignment machine, such as that shown in Figure 17.

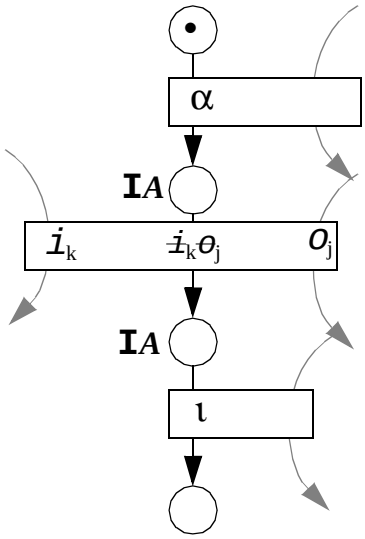


Figure 17 Typed Petri Net Showing an Assignment Machine $=^{j\to k}$

Other operations on basic data types may also be modeled as interaction machines. For example, relations may be modeled as interaction machines that read values from two value machines and set the value of a boolean value machine to indicate whether the first two values belong to the relation. Note that this is an operational definition of a relation: while it does implicitly define the relation itself, it actually tests membership in the relation. Arithmetic and boolean operations are other examples of operations that may be modeled as interaction machines.

3.2.1 Interaction Machines Model Primitive Operations

It is important to note that interaction machines are not the only means of modeling these operations, but the alternative means of implementing these operations are behaviorally distinct from (not behaviorally equivalent to) interaction machine models. For example, the

assignment operation could be alternately modeled by copying the value from one machine into a second machine (using an interaction machine), and from there copying the value into the real destination machine (using another interaction machine). If the variables involved kept track of the number of accesses that had occurred, then an observer would be able to distinguish the two assignment operations because the second option could reach a state in which the access counts on the variables are different, while the interaction machine model could never reach such a state. Deciding which operations are primitive in this sense is an important issue in modeling languages that will generate code for parallel machines.

3.3 Activation Machines

Thus far we have defined machines that store values and machines that facilitate the exchange of values between value machines. We now address the issue of coordinating the activation and idling of these machines. To accomplish this, we define a special kind of interaction machine that is labeled exclusively with the inverses of the activate and idle actions of other machines and possibly an activation and/or idle action of its own. We shall call these machines ***activation machines***. We shall place some further restrictions upon the appearance of activation and idle actions: the inverses of activate and idle actions may only appear on activation machines, and then on no more than one machine. As we shall see shortly, the syntax will make clear exactly which machine these action inverses appear on.

As an example, let us suppose that we have two machines, $\mathbf{M}_i$ and $\mathbf{M}_j$, and we wish to have $\mathbf{M}_i$ go idle while activating $\mathbf{M}_j$. To accomplish this, we compose these two machines with an activation machine $\mathbf{A}^{i\to j}$ that responds to both the idle action of $\mathbf{M}_i$ and the activate action of $\mathbf{M}_j$. This machine is defined by the following equation:

$$\mathbf{A}^{i\to j} \equiv A_0 \equiv \overline{\iota_i}\,\overline{\alpha_j}.A_0$$

This kind of recursive expression arises frequently, and we will often want to give the machine as a term in a state algebra expression without having to define a constant such as A_0 in the above expression. Accordingly, we will use the notation:

$$(\bar{\iota}_i\bar{\alpha}_j.)^*$$

where $(\bar{\iota}_i\bar{\alpha}_j.)^*$ is a shorthand notation for $\bar{\iota}_i\bar{\alpha}_j.\bar{\iota}_i\bar{\alpha}_j.\bar{\iota}_i\bar{\alpha}_j$..... This machine is shown in Figure 18.:

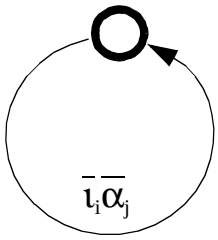


Figure 18 Activation Machine $A^{i\to j}$

While we could use our original notation and write the composition as:

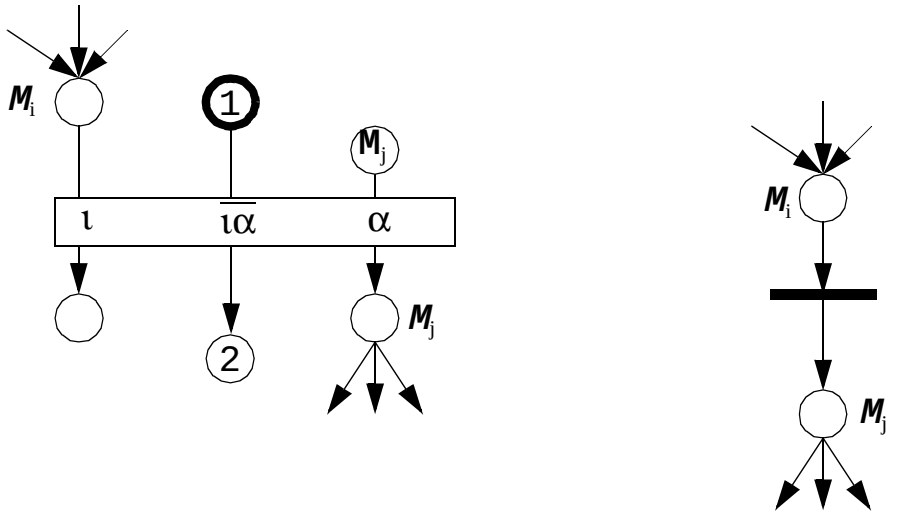
$$(\mathbf{M}_i \mid \mathbf{A}^{i\to j} \mid \mathbf{M}_j)\backslash\{\iota_i,\alpha_j\}$$

this situation arises so frequently that we will use a shorthand notation as follows:

$$\mathbf{M}_i;\, \mathbf{M}_j \equiv (\mathbf{M}_i \mid \mathbf{A}^{i\to j} \mid \mathbf{M}_j)\backslash\{\iota_i,\alpha_j\}$$

Here the semicolon indicates that the machines before and after are composed with an activation machine that relates the idling of the machine before the semicolon to the activation of the machine after the semicolon. The semicolon is associative (Appendix B Theorem 26).

Just as we have removed the explicit representation of the activation machine from our equational syntax, we also remove it from our reduced Petri net notation. Figure 19 shows a partial reduced Petri net for both the unsimplified and simplified versions.



a) without simplification *a) with simplification*

Figure 19 Reduced Petri Net Fragment for Composition $M_i \mid A^{i\to j} \mid M_j$

Another common activation construct is the simultaneous activation or termination of multiple machines. We might wish, for example, to activate machines M_j and M_k in parallel, activating them as machine M_a terminates, and not starting machine M_b until both have terminated. This would require two activation machines, $A^{a\to jk}$ relating the idling of machine M_a to the activation of machines M_j and M_k, and $A^{jk\to b}$ relating the idling of machines M_j and M_k to the activation of machine M_b. Machine $A^{a\to jk}$ is defined by the equation:

$$A^{a\to jk} \equiv (\bar{\iota}_a \bar{\alpha}_j \bar{\alpha}_k .)^*$$

and is shown in Figure 20.

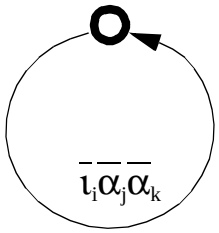


Figure 20 Activation Machine $\mathbf{A}^{i\to jk}$

Again, while we could use our original notation and give the full composition of machines as:

$$((\mathbf{M}_a \mid \mathbf{A}^{a\to jk} \mid \mathbf{M}_j \mid \mathbf{M}_k)\backslash\{\iota_a,\alpha_j,\alpha_k\} \mid \mathbf{A}^{jk\to b} \mid \mathbf{M}_b)\backslash\{\iota_j,\iota_k,\alpha_b\}$$

this situation arises frequently enough to warrant its own notation. We first introduce a notation to indicate that two or more machines should be activated and idled in parallel:

$$\langle \mathbf{M}_1 \mid \mathbf{M}_2 \mid ... \mid \mathbf{M}_n \rangle = ((\alpha_x\overline{\alpha_1}\overline{\alpha_2}...\overline{\alpha_n}.)^* \mid \mathbf{M}_1 \mid \mathbf{M}_2 \mid ... \mid \mathbf{M}_n \mid (\iota_x\overline{\iota_1}\overline{\iota_2}...\overline{\iota_n}.)^*)\backslash\{\alpha_1, \alpha_2,...,\alpha_n,\iota_1,\iota_2,...,\iota_n\}$$

Note that this notation introduces new activate and idle actions (α_x and ι_x) for the composition. We may then use the semicolon notation as before:

$$\mathbf{M}_a;\ \langle \mathbf{M}_j \mid \mathbf{M}_k \rangle;\ \mathbf{M}_b$$

Figure 21 shows the partial typed Petri net for this composition. The composition is equivalent to:

$$\mathbf{M}_a \mid (\overline{\iota_a}\overline{\alpha_x}.)^* \mid (\alpha_x\overline{\alpha_j}\overline{\alpha_k}.)^* \mid \mathbf{M}_j \mid \mathbf{M}_k \mid (\iota_x\overline{\iota_j}\overline{\iota_k}.)^* \mid (\overline{\iota_x}\overline{\alpha_b}.)^* \mid \mathbf{M}_b$$

where we have omitted the restrictions for clarity. Theorem 25 (Appendix B) can be used to show that $((\bar{\iota}_a\bar{\alpha}_x.)^* \mid (\alpha_x\bar{\alpha}_j\bar{\alpha}_k.)^*)\backslash\{\alpha_x\} = (\bar{\iota}_a\bar{\alpha}_j\bar{\alpha}_k.)^*$, which means that the syntax "; ⟨" gives us the desired relationship between the idling of $\mathbf{M}_a$ and the activation of $\mathbf{M}_j$ and $\mathbf{M}_k$.

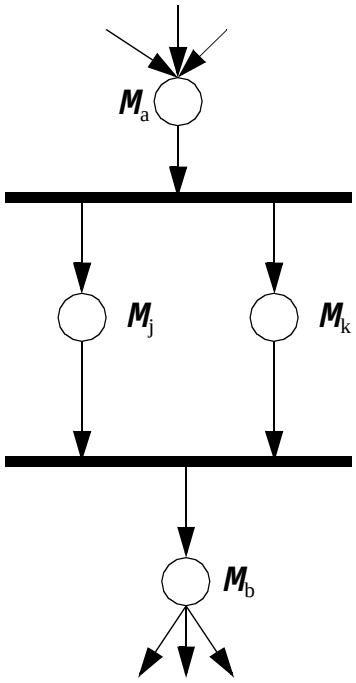


Figure 21 Partial Typed Petri Net of Composition $\mathbf{M}_i$; $\langle \mathbf{M}_j \mid \mathbf{M}_k \rangle$; $\mathbf{M}_l$

3.4 Machine Algebra

We define a ***well-behaved machine*** to be any machine that has exactly one observable activate action and whose initial action is always that unique activate action, and has exactly one observable idle action and if this idle action occurs, the only observable action that may follow is the activate action[1].

We note that ";" and "⟨⟩" both preserve the well-behaved property:

1. Note that this does not imply that the machine must respond to the idle action in every state. This is similar to Milner's well-terminating property [Miln89] p. 173

Lemma 3

Let M be the set of well-behaved machines. Then:

$$\mathbf{M}_1, \mathbf{M}_2 \in M \Rightarrow \mathbf{M}_1;\mathbf{M}_2 \in M$$

$$\mathbf{M}_1, \mathbf{M}_2,...,\mathbf{M}_n \in M \Rightarrow \langle \mathbf{M}_1 | \mathbf{M}_2 | ... | \mathbf{M}_n \rangle \in M$$

We adopt the convention that the ";" operator binds more tightly than (takes precedence over) the | operator, and that $\langle\rangle$ binds more tightly than either.

Our ***machine algebra*** is then the system (M, =, ;, |, $\langle\rangle$), where M is the set of well-behaved machines, = is behavioral equivalence, and ;, |, and $\langle\rangle$ are as defined above.

3.5 A simple example

Let us now construct a simple model of a single instance of the following program fragment:

```
declare
   a : integer := 2;
   b : integer := 3;
   c : integer;
begin
   c := a + b;
end;
```

Example 1 Simple Program Fragment

If all of the declarations were to occur in parallel (which is not true in Ada) the program fragment could translate into the following semantic expression, which is illustrated in Figure 22:

$$\langle \mathbf{V}_a \mid \mathbf{V}_b \mid \mathbf{V}_c \mid \mathbf{C}_2 \mid \mathbf{C}_3 \mid =_1^{2\to a} ;=_2^{3\to b} ;\langle \mathbf{V}_t \mid +^{a,b\to t};=_3^{t\to c} \rangle\rangle \tag{12}$$

Note that the three value machines and the two constant machines are started in parallel, along with the machine $(=_1^{2\to a} ;=_2^{3\to b} ;\langle \mathbf{V}_t \mid +^{a,b\to t};=_3^{t\to c}\rangle)$. This serial combination first initializes $\mathbf{V}_a$, then initializes $\mathbf{V}_b$ and then starts the machine parallel combination of a temporary value machine $\mathbf{V}_t$ and the series combination of the addition operation followed by the assignment to $\mathbf{V}_c$.

It is reasonable to ask why the temporary variable $\mathbf{V}_t$ is constrained to have the lifetime of the two statements that interact with it. Why not use the following expression, in which the temporary variable has the lifetime of the program:

$$\langle \mathbf{V}_a \mid \mathbf{V}_b \mid \mathbf{V}_c \mid \mathbf{C}_2 \mid \mathbf{C}_3 \mid \mathbf{V}_t \mid =_1^{2\to a} ;=_2^{3\to b} ; +^{a,b\to t};=_3^{t\to c} \rangle \tag{13}$$

The answer is that these two expressions are indeed equivalent, subject to a restriction on the accessibility of $\mathbf{V}_t$. We will now proceed to state a necessary theorem and then give the derivation of (13) from (12).

3.5.1 Parallelization Theorem

We wish to consider the circumstances under which the sequencing of machines in a composition may be altered. Consider the following abstracted model of a program or subprogram:

$$\langle \mathbf{V}_a \mid \mathbf{V}_b \mid \mathbf{M}_p; \mathbf{M}_q ; \mathbf{M}_r ; \mathbf{M}_s\rangle \backslash S$$

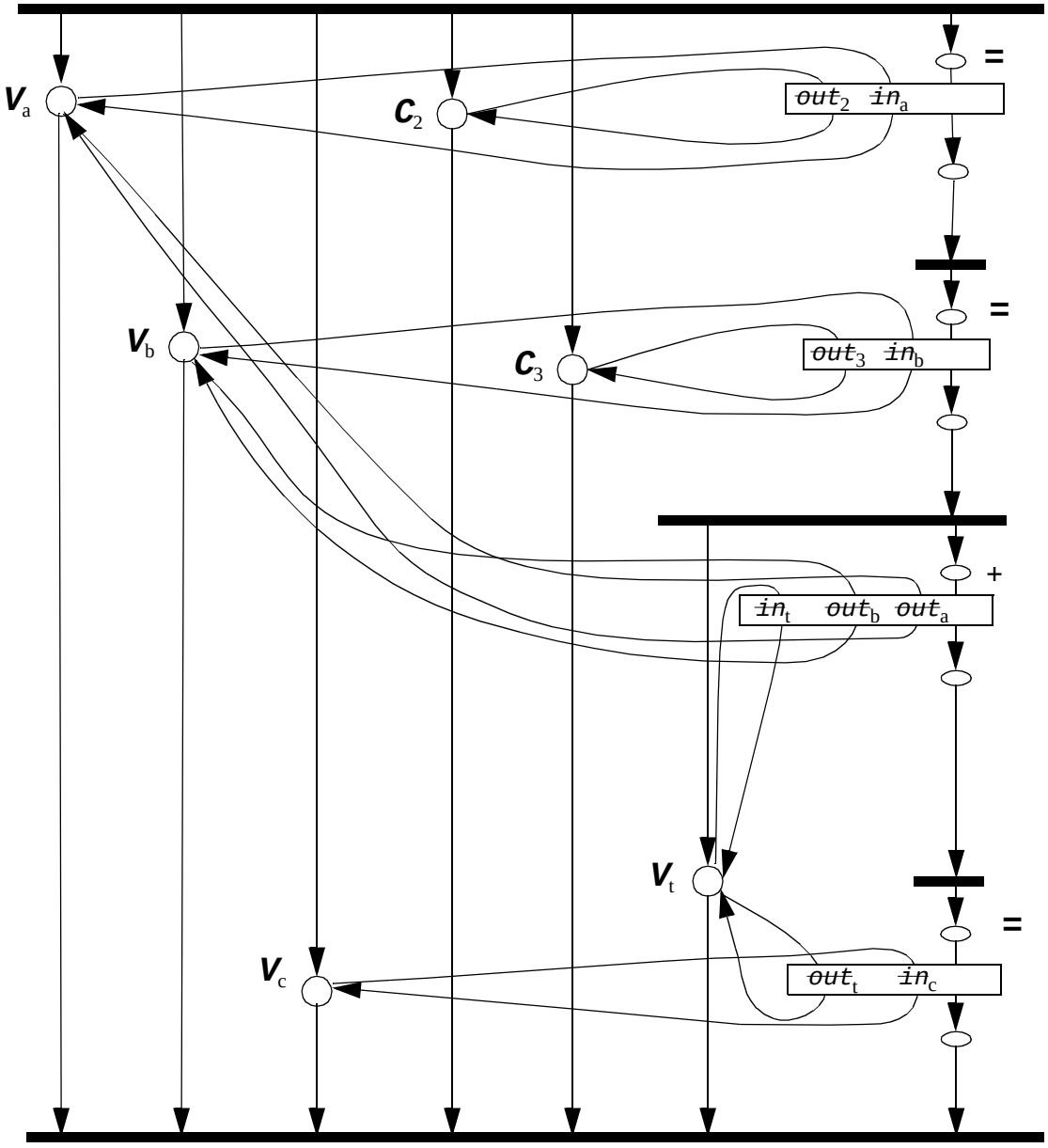


Figure 22 Reduced Petri Net of Example 1

where $S = Sort(\mathbf{M}_a) \cup \overline{Sort}(\mathbf{M}_a) \cup Sort(\mathbf{M}_b) \cup \overline{Sort}(\mathbf{M}_b) \cup Sort(\mathbf{M}_q) \cup \overline{Sort}(\mathbf{M}_q) \cup Sort(\mathbf{M}_r) \cup \overline{Sort}(\mathbf{M}_r)$, $\mathbf{V}_a$ and $\mathbf{V}_b$ are local variables of the program, $\mathbf{M}_p$ copies actual parameter values into the local variables, $\mathbf{M}_q$ and $\mathbf{M}_r$ are machines that do the actual work of the subprogram by modifying the local variables, and $\mathbf{M}_s$ copies the final values out into their target destinations. The restriction $\backslash S$ simply says that the internal workings of the program are not visible from outside the program (the variables are hidden and the machines that do the work cannot communicate with any machines outside of the program). The sequencing $\mathbf{M}_p ; \mathbf{M}_q ; \mathbf{M}_r ; \mathbf{M}_s$ says that the actual parameter values are copied in, then $\mathbf{M}_q$ does its work, then $\mathbf{M}_r$ does its work, and finally $\mathbf{M}_s$ copies the results back out of the program.

Now it seems intuitive that if the variables cannot interact with each other, and the machines that do the work ($\mathbf{M}_q$ and $\mathbf{M}_r$) cannot interact with each other, and each working machine can only interact with one of the variables, then it should not matter which order $\mathbf{M}_q$ and $\mathbf{M}_r$ do their work. In fact, they could even operate in parallel! This is exactly what the parallelization theorem establishes:

Theorem 4 Parallelization Theorem

$$\mathbf{M}_a \perp \mathbf{M}_b, \mathbf{M}_a \perp \mathbf{M}_r, \mathbf{M}_b \perp \mathbf{M}_q, \mathbf{M}_q \perp \mathbf{M}_r \Rightarrow$$
$$\langle \mathbf{M}_a \mid \mathbf{M}_b \mid \mathbf{M}_p ; \mathbf{M}_q ; \mathbf{M}_r ; \mathbf{M}_s \rangle \backslash S =$$
$$\langle \mathbf{M}_a \mid \mathbf{M}_b \mid \mathbf{M}_p ; \langle \mathbf{M}_q \mid \mathbf{M}_r \rangle ; \mathbf{M}_s \rangle \backslash S =$$
$$\langle \mathbf{M}_a \mid \mathbf{M}_b \mid \mathbf{M}_p ; \mathbf{M}_r ; \mathbf{M}_q ; \mathbf{M}_s \rangle \backslash S$$

This is a valuable result, since it shows how to take a serial program and convert it into an equivalent parallel program with no analysis beyond simply determining the orthogonality

(independence) of the component parts of the program. The proof of this theorem is in Appendix B section B.1 on page 160.

3.5.2 Change of Scope Theorem

Somewhat similar to the parallelization problem is the related change of scope theorem. Consider the following configuration of machines:

$$\langle \mathbf{M}_a ; \langle \mathbf{M}_b \mid \mathbf{V}_x \rangle ; \mathbf{M}_c \rangle \backslash S$$

where $S = Sort(\mathbf{V}_x)$. The situation we are modeling here is one in which $\mathbf{V}_x$ is a local variable used by $\mathbf{M}_b$ only. If $\mathbf{M}_a$ and $\mathbf{M}_c$ cannot interact with $\mathbf{V}_x$ and $\mathbf{V}_x$ is hidden from the outside, then it seems reasonable that the lifetime of $\mathbf{V}_x$ could be extended, yielding a behaviorally equivalent configuration:

$$\langle \mathbf{V}_x \mid \mathbf{M}_a ; \mathbf{M}_b ; \mathbf{M}_c \rangle \backslash S$$

This leads to the following theorem:

Theorem 5 Change of Scope Theorem

$$\mathbf{M}_a \perp \mathbf{M}_x, \mathbf{M}_c \perp \mathbf{M}_x \Rightarrow$$
$$\langle \mathbf{M}_a ; \langle \mathbf{M}_b \mid \mathbf{M}_x \rangle ; \mathbf{M}_c \rangle \backslash S = \langle \mathbf{M}_x \mid \mathbf{M}_a ; \mathbf{M}_b ; \mathbf{M}_c \rangle \backslash S$$

where $S = Sort(\mathbf{M}_X)$. The proof of this theorem is in Appendix B section B.2 on page 168.

We now return to our example of (12). We now wish to show that the scope of the local variable $\mathbf{V}_t$ can be changed without affecting the observable behavior of the expression. This example actually contains an omission: the temporary variable $\mathbf{V}_t$ is not visible outside

of the assignment statement. Accordingly, we add the restriction hiding the existence of the temporary variable $\mathbf{V}$:

$$\langle \mathbf{V}_a \mid \mathbf{V}_b \mid \mathbf{V}_c \mid \mathbf{C}_2 \mid \mathbf{C}_3 \mid =_1^{2\to a} ;=_2^{3\to b} ;\langle \mathbf{V}_t \mid +^{a,b\to t};=_3^{t\to c} \rangle \backslash \mathit{Sort}(\mathbf{V}_t)\rangle$$

Next, we must show that some of the machines are orthogonal. By definition, all of the variables and constants are orthogonal. We also note that $=_1^{2\to a}$ has only the actions of $\mathbf{C}_2$ and $\mathbf{V}_a$, and $=_2^{3\to b}$ has only the actions of $\mathbf{C}_3$ and $\mathbf{V}_b$. We thus have:

$$\mathbf{V}_t \perp =_1^{2\to a} \text{ and } \mathbf{V}_t \perp =_2^{3\to b}$$

These orthogonality observations, in conjunction with Appendix A (65) and the definition of ; allow us to change the scope of restriction:

$$\langle \mathbf{V}_a \mid \mathbf{V}_b \mid \mathbf{V}_c \mid \mathbf{C}_2 \mid \mathbf{C}_3 \mid (=_1^{2\to a} ;=_2^{3\to b} ;\langle \mathbf{V}_t \mid +^{a,b\to t};=_3^{t\to c} \rangle) \backslash \mathit{Sort}(\mathbf{V}_t)\rangle$$

Now we apply Theorem 5, letting $\mathbf{M}_a = =_1^{2\to a} ;=_2^{3\to b}$, $\mathbf{M}_b = +^{a,b\to t};=_3^{t\to c}$, $\mathbf{M}_x=\mathbf{V}_t$ and $\mathbf{M}_c=\mathbf{0}$ to get:

$$\langle \mathbf{V}_a \mid \mathbf{V}_b \mid \mathbf{V}_c \mid \mathbf{C}_2 \mid \mathbf{C}_3 \mid (\mathbf{V}_t \mid =_1^{2\to a} ;=_2^{3\to b} ;+^{a,b\to t};=_3^{t\to c}) \backslash \mathit{Sort}(\mathbf{V}_t)\rangle$$

Again applying our orthogonality observations and Appendix A (65) we get our desired result:

$$\langle \mathbf{V}_a \mid \mathbf{V}_b \mid \mathbf{V}_c \mid \mathbf{C}_2 \mid \mathbf{C}_3 \mid \mathbf{V}_t \mid =_1^{2\to a} ;=_2^{3\to b} ;+^{a,b\to t};=_3^{t\to c} \rangle \backslash \mathit{Sort}(\mathbf{V}_t)$$

4.0 Visibility

Determining the visibility of variables, functions and types at various points in a program can be a difficult and complicated task, especially in languages as complex as Ada. While intuition will successfully guide a programmer in a simple block structured language, concepts such as separate specifications and implementation in Ada and Modula-2, and **use** clauses in Ada make it increasingly difficult for the programmer to build and apply a conceptually simple model of visibility.

In this chapter we examine the process of determining visibility, and propose the use of a modified set union known as a masking union as a means of formalizing visibility rules. In contrast with the traditional ad-hoc rule based approach to defining visibility, this approach has the advantage of being both conceptually simple and mathematically rigorous. We begin by using masking unions to model visibility in simple block structures, and then proceed to analyze increasingly complex visibility computations, including considerations of declaration ordering and inter-dependency. We observe that the resulting models have the additional advantage of providing diagnostic information that can be used to explain, in a meaningful way, why certain program elements are not visible at specific places in the program. In [Brow90] we have considered the more complex visibility problems associated with the separation of specifications and implementations in Ada, and provide an analysis of library-level visibility rules in.

While all examples are drawn from the Ada programming language, no knowledge of the Ada language is assumed. Initial examples simply use blocks with declarations, which should be readily understandable to readers not familiar with Ada. As other Ada features are introduced, they are defined and explained.

4.1 Declarations and References

In the previous chapters we have laid the groundwork for modeling the semantics of a programming language in terms of machines. The meaning of an identifier in a program is a machine. We define an ***environment*** $E{:}Ide{\times}M$ to be a relation between identifiers and machines. We note that this environment is suitable for mapping identifiers into both machine instances and types if we use the "defining machine" approach to machine types.

A ***declaration*** is an entry in an environment relation, mapping an identifier to a machine[1]. A ***reference*** is the use of an identifier to locate the appropriate declaration in an environment relation and thus locate the machine that is being referenced[2]. *Each occurrence of an identifier in a program is either part of a new declaration or it is part of a reference*. Consider the program in Example 2. In this example, we observe a single explicit declaration[3] of a variable named `a`, and a number of references: one to `integer` in the declaration itself; another to `a` in the assignment statement; and a third to the literal `1` in the assignment statement. As we shall see later when we complete our semantics, `:=` will itself be viewed as a reference to an assignment operator.

```
X: declare
      a : integer; -- a declaration of "a"
   begin
      a := 1; -- a reference to the "a"
   end X;
```

Example 2 Declarations and References

1. Note that this allows more than one identifier to be associated with the same machine.
2. As we shall soon see, there may be more than one entry in the relation with the same identifier
3. Explicit in the sense that there is an explicit declaration in the code. We shall see later that some machines are implicitly declared

Taking this perspective of a program, an interesting problem is the determination of which declaration a particular reference actually refers to. This problem can, in itself, be divided into two sub-problems: one is the determination of which declarations are *visible* at a given point in the program (we call this set of visible declarations the ***direct environment***); the other is, of the declarations visible at this point, which is the one that is actually being referred to. In this thesis we will confine our investigation to the computation of visibility, and leave the formalization of how a reference is selected for future work.

4.2 Homographs and Overload Resolution

Most programming languages place some restrictions on the declarations that are allowed to co-exist in the same direct environment. Most languages do not allow the program of Example 3, in which two variables of the same name, `a`, but of different types, appear in block `X`. Similarly, the dual declarations of `b` are not allowed. Note that despite the syntactic identity of the two declarations of `b`, they are two distinct declarations that happen to associate two different machines of the same type with the same identifier. Maintaining these separate declarations is an essential part of the visibility model that we will be developing.

```
X: declare
      a : integer;
      a : boolean;
      b : boolean;
      b : boolean;
   begin
      b := true; -- an ambiguous reference to “b”
   end X;
```

Example 3 Ambiguous References

On the other hand, most languages support the notion of names being re-used in nested blocks, as in Example 4.

```
X: declare
      a : integer;
   begin
Y:    declare
         a : boolean;
      begin
         a := 1; -- which declaration does this refer to?
      end;
   end;
```

Example 4 Re-use of Names

Under certain circumstances, languages may allow the re-use of a name within the same direct environment, as in Example 5. In this example, two functions are declared, both with the name `f`. When a situation like this occurs, we say that the name is ***overloaded***, or has more than one declaration associated with it. When a reference to `f` occurs, the correct machine must be selected from among the declarations associating machines with `f`. This process of selecting the actually referenced machine is known as ***overload resolution***. We shall leave the formalization of overload resolution in our model an open problem in this thesis.

```
X: declare
      a : integer;
      function f return integer is
      begin
         return 1;
      end f;
      function f return boolean is
```

```
   begin
      return true;
   end f;
begin
   a := f; -- overload resolution needed to select "f"
end;
```

Example 5 Overload Resolution

How do we characterize declarations so that we know when overloading is allowed and when it is not? The answer is based upon the manner in which the declarations can be distinguished from each other. If two declarations, d_1 and d_2, are never distinguishable, based upon some set of information about the declarations, we call them ***homographs***, and define a corresponding relation $\boldsymbol{H}(d_1, d_2)$. We note that the homograph relation may be an arbitrary relation, but we also note that an understanding of the structure of this relation is an essential part of understanding visibility. Thus it is in the interest of both the language designer and the practicing programmer to keep the rules for defining this relation as simple as possible.

While the exact definition of homograph will vary from language to language, we assume that such a definition can be given for each language. ***In languages that do not allow overloading, two declarations are homographs if they share the same name.*** In languages that do allow overloading, generally overloading is only allowed for machines that play certain roles in the language. In Ada, for example, overloading is allowed for machines used as subprograms, but not for machines used as variables. The implication is that declarations in such languages must be extended to contain information about the role that the machine is playing. In such languages we would need to use an ***extended environment***

E:$Ide \times M \times Role$ where *Role* indicates the role that the machine plays in the language (variable, function, type, etc.).

Languages frequently differ in the distinguishability criteria that they use, and hence differ in their definitions of homograph. Some languages, like Ada, even use different criteria in different situations within the same program! For example, Ada does not allow subprograms that have the same parameter and return type profile to be *declared* in the same context, but it does allow two such declarations to be *brought into the same environment* via **use** clauses. Furthermore, the overload resolution process, which *selects* the appropriate declaration, is allowed to also use any formal parameter names that happen to be used in the subprogram call to differentiate and select a single declaration.

As a result, in order to fully model Ada visibility, we shall need three homograph definitions: the main one, which will be used in most visibility computations, considers subprograms to be homographs if their names and parameter and result type profiles match (their associated machines have the same state type); the second, which we shall use for defining **use** clause semantics, considers all subprograms to be differentiable, even if they have the same name and type profiles; and the third, which is only required to describe the overload resolution process, considers subprograms to be homographs if their names, parameter and result type profiles *and formal parameter names* match.

We thus have the following basic definition of an ***Ada Homograph*** [LRM 8.3(15)] for use in visibility computations (here we formalize distinguishability of declarations *declared* in the same environment - the second variant of the homograph declaration will be provided in a later chapter for the **use** clause case). Two declarations are ***Ada Homographs*** of each other if they have the same identifier and either:

1 Overloading is not allowed for at least one of the pair (based on roles); or

2 Overloading is allowed for both, but they are indistinguishable on the basis of parameter and return types.

We note that the homograph definition for overload resolution (which we have not formally given) will require the names of the parameters, and that this information is not present even in our extended environment relation. We thus define an ***Ada environment*** E:$Ide \times M \times Role \times ParameterNameList$ where *ParameterNameList* is simply an ordered list of formal parameter names.

We shall, in the remainder of this thesis, use this Ada definition of homograph in our examples. Since we are not going to go into the details of overload resolution, we shall omit the *ParameterNameList* from our formal descriptions of environments. Languages with other characteristics can be modeled by suitably changing the definition of homograph and environment.

We note that the relation defined by the Ada definition of homograph is reflexive and symmetric, but is *not* transitive. Consider the following Ada declarations:

```
a : integer;
procedure a (b:integer);
procedure a (b:boolean);
```

The first declaration is a homograph of both the second and the third, but the two procedure declarations are not homographs of each other. If we were to relax the restriction that overloading is only allowed for some types of declarations, and allow all declarations to be overloadable, then we would get a homograph definition that leads to an equivalence rela-

tion, i.e. a relation that is reflexive, transitive, and symmetric, and a corresponding simplification in understanding visibility.

4.3 Environments and the Masking Union

We now return to our primary purpose, which is to describe the visibility of declarations (associations of names with machines) at various points in the program. In a typical programming language we will find basic groupings of declarations and executable code, which we shall call a ***declarative block***. ***Each declarative block is actually the definition of a machine in our model*** (usually the prototype machine of a state type). This machine is actually just a composition of the machines defined by its declarations and executable statements. The defined machine is then associated with its name and placed in an environment for later reference. The machines defined by the declarations will themselves be placed in environments, either for reference within this machine or, in some cases, for reference by other machines.

There are at least two environments that are of major interest with respect to a declarative block: the set of declarations that are visible in the executable body of the block, known as the ***direct environment*** (**DE**); and the set of declarations that occurred within the block, known as the ***local declarations*** (**LD**).

There are, as we shall soon see, a number of environments in addition to these two associated with each declarative block. We formalize the relationship between declarative blocks ad environments by defining a ***scope*** $S{:}M{\times}E{\times}Role$ to be a relation associating machines M with environments E and the roles that the environments play with respect to those machines. We shall soon see that defining visibility is basically the process of defining these environments and their relationships with machines. We shall use the notation **M**:**R** to des-

ignate the environments playing role **R** with respect to machine **M**. Thus **X**:**DE** would be the direct environment associated with machine **X**.

Consider the nested blocks of Example 6, and the task of determining block **Y**'s direct environment. If we consider block **Y**'s local declarations **Y**:**LD**, and block **X**'s direct environment **X**:**DE**, one might be tempted to conclude that block **Y**'s direct environment **Y**:**DE** is simply the union of block **Y**'s local declarations and block **X**'s direct environment:

$$\mathbf{Y}\text{:}\mathbf{DE} = \mathbf{Y}\text{:}\mathbf{LD} \cup \mathbf{X}\text{:}\mathbf{DE} \tag{14}$$

```
X: declare
      a : integer;
      b : boolean;
      c : integer;
   begin
Y:    declare
         d : boolean;
         e : integer;
         f : boolean;
      begin
         a := 1;
      end Y;
   end Y;
```

Example 6 Basic Nested Blocks

While the union provides a useful approximation to what is actually visible in block **Y**, it does not provide the correct result for Example 4, in which only the declaration of a that occurred in block **Y** is visible to the assignment statement: the declaration of a that occurred

in block **X** is hidden. With a simple set union, both would be visible. We thus need to modify our notion of set union in order to adequately model visibility computations.

We now proceed to define a variation of a set union known as a ***masking union***, so called because declarations that occur in one of its source sets will "mask" the presence of declarations occurring in the other source set. We shall use the symbol $\underset{m}{\cup}$ to designate this operator.

Given two sets, A and B, the masking union of these sets

$$C = A \underset{m}{\cup} B$$

is defined by:

$$C = \{a \in A\} \cup \{b \in B | \forall a \in A \neg \boldsymbol{H}(a, b)\} \quad (15)$$

where ***H*** is the homograph relation defined in section 4.2 on page 70.

Appendix 1 describes some of the properties of this operator and some basic theorems.

4.4 Basic Visibility Computations

Utilizing masking unions, we can now complete the model of visibility computations in our simple block structure. For each declarative block, we have

$$\mathbf{X}\text{:}\mathbf{DE} = \mathbf{X}\text{:}\mathbf{LD} \underset{m}{\cup} \mathbf{X}^{-1}\text{:}\mathbf{DE} \quad (16)$$

where $\mathbf{X}^{-1}$ denotes the "parent" of **X**, i.e. the declarative block in which the declaration of **X** occurred.

For example, consider the program fragment of Example 7:

```
X: declare
      a : integer;
      d : integer;
   begin
Y:    declare
         a : integer;
         b : integer;
      begin
Z:       declare
            b : integer;
            c : integer;
         begin
         end Z;
      end Y;
   end X;
```

Example 7 Nested Blocks

We first note that the keyword **`declare`** marks the beginning of each new declarative region, and the region ends with the keyword **`end`**. The keyword **`begin`** simply serves to separate the declarations from the executable portion of the block. It is easy to see that the direct environments of the blocks are as follows (omitting the roles and parameter names from the relations, and ignoring pre-defined declarations other than integer):

$\mathbf{X}^{-1}$:**DE** = {(X,**X**), (integer,**Int**)}

X:**DE** = {(a,**a**∈**X**), (d,**d**∈**X**), (X,**X**), (Y,**Y**), (integer,**Int**)}

Y:**DE** = {(a,**a**∈**Y**), (b,**b**∈**Y**), (d,**d**∈**X**), (X,**X**), (Y,**Y**), (Z,**Z**), (integer,**Int**)}

$$\mathbf{Z}\text{:}\mathbf{DE} = \{(\texttt{a},\mathbf{a}\in\mathbf{Y}), (\texttt{b},\mathbf{b}\in\mathbf{Z}), (\texttt{c},\mathbf{c}\in\mathbf{Z}), (\texttt{d},\mathbf{d}\in\mathbf{X}), (\texttt{X},\mathbf{X}), (\texttt{Y},\mathbf{Y}), (\texttt{Z},\mathbf{Z}),$$
$$(\texttt{integer},\mathbf{Int})\}$$

where we use the notation $\mathbf{a}\in\mathbf{X}$ to disambiguate **a**, indicating that we want the **a** that is a part of **X**. The notion of "being a part of **X**" will become clear in the next chapter. For now, it is sufficient to think of it as meaning "was declared in **X**." $\mathbf{X}^{-1}$:**DE** represents the direct environment of the block in which X was declared.

4.4.1 Compilation Diagnostic Aids

It is not unusual for a programming language to have a restriction that two declarations occurring in the same declarative region are not allowed to be homographs of each other. Our model is flexible enough to allow homographs to be declared in the same declarative region (recall that these declarations always maintain their separate identities). In addition, our model provides a convenient means of checking to see if the rule regarding homographs has been violated for a declaration $d \in \mathbf{LD}$:

$$\forall d' \in \mathbf{LD}, d \neq d', \neg \boldsymbol{H}(d, d')$$

If this term is false, then there is a violation of the restriction. If we assume a constant comparison time, then the worst case complexity of this operation is $O(n^2)$, where n is the number of declarations. However, if we consider that, for most definitions of homograph, name equivalence is a major part of the homograph predicate (declarations with different names will never be homographs of each other), the average complexity of the check can be reduced by indexing **LD** by the names of the declarations, and only applying the homograph check to those declarations with the same name. This yields an average complexity of $O\left(n\log\left(\frac{n}{m}\right) + (n \times m)\right)$, where m is the average number of declarations with the same

name, and n is the total number of declarations in the set. If hashing is used, the first term reduces to a constant, and the resulting complexity is $O(n \times m)$ Not only does this give a boolean result, but it points out specifically which declarations are homographs of the current declaration, thus providing the basis for meaningful compilation diagnostic messages.

4.5 Visibility and the Ordering of Declarations

The ordering of the declarations within a declarative region can have an effect upon the meaning of references within a declarative region. In the following we examine three possible semantics for visibility within the declarative regions: letrec (let recursive), let and let*.

4.5.1 Letrec (Let Recursive) Visibility Semantics

Letrec, or let recursive, semantics indicates that any declaration in a context may refer to any other declaration in the region, including itself – hence the reference to recursion. This was the semantics given in Section 4.4, where we noted that the ordering of declarations within a block was irrelevant. With this semantics, the programs of Example 8 and Example 9 have identical semantics, with **Y**:**DE** = {(a,**a**∈**Y**), (b,**b**∈**Y**), (X,**X**), (Y,**Y**), (integer,**Int**)} in both cases, and all references within the block **Y** seek to locate their declarations in this environment. This is the semantics of the scheme `letrec` construct [Rees86]. All of the declarations in a given block are added to the environment before any of the references are evaluated, and any declaration is allowed to reference any other declaration, including itself. Thus, in both examples, the variable `b` is initialized to the value of 4.

```
X: declare
       a : integer := 3;
   begin
```

```
Y:    declare
         b : integer := a; -- which "a" does this refer to?
         a : integer := 4;
      begin
         a := 1;
      end Y;
   end X;
```

Example 8 Relative Ordering of Declarations

```
X: declare
      a : integer := 3;
   begin
Y:    declare
         a : integer := 4;
         b : integer := a; -- which "a" does this refer to?
      begin
         a := 1;
      end Y;
   end X;
```

Example 9 Relative Ordering of Declarations

4.5.2 Let Visibility Semantics

A second alternative for visibility semantics is that associated with the `let` construct in Lisp [Stee84] and Scheme. In this semantics the direct environment computation is the same as for the `letrec` construct, but references that occur *as part of declarations* seek to locate the referenced declarations in the direct environment of the next outer construct (**X**:**DE** for declarations in **Y**), while references that occur *after the declarative region* (in the set of statements) locate their referenced declarations in the direct environment of the block in which

they occur (**Y**:**DE** for statements in Y). With this semantics, the declarations of `b` in both examples would be initialized to the value of 3.

4.5.3 Let* Visibility Semantics

A third alternative is the visibility semantics associated with the `let*` constructs in Lisp and Scheme. In this semantics, each declaration introduces a new direct environment. The references that occur in a given declaration locate the referenced items in the direct environment of the previous declaration (or some initial direct environment $\mathbf{DE}_0$, if this is the first declaration in the block). References within the set of statements locate their referenced declarations in the direct environment associated with the last declaration. With this semantics, the declaration of `b` in Example 8 would be initialized to the value of 3, whereas the declaration of `b` in Example 9 would be initialized to the value of 4.

The direct environment associated with each declaration can be more formally described as follows. Given:

a an ordered set **LD** of k declarations $\{d_1, d_2, ..., d_k\}$

b an initial direct environment $\mathbf{DE}_0$

We might be tempted to define the direct environment associated with each declaration as:

$$\mathbf{DE}_i = \{d_i\} \underset{m}{\cup} \mathbf{DE}_{i-1} \tag{17}$$

We would then like to be able to make the simplifying assumption that

$$\mathbf{DE} = \mathbf{DE}_k = \mathbf{LD} \underset{m}{\cup} \mathbf{DE}_0 \tag{18}$$

where k is the last declaration. Unfortunately, this is not the case: if $\boldsymbol{H}(d_i, d_j)$ for some $i < \varphi$, then $\mathbf{DE}_k$ will contain d_j but not d_i if computed using (17), and will contain both if computed using (18).

For a language that does not allow homographs to be declared in the same declarative region, (18) gives the correct result for correct programs. But to accommodate languages whose semantics allow homographs in **LD**, we provide a different formulation of the direct environment $\mathbf{DE}_i$. Given:

a an ordered set **LD** of k declarations $\{d_1, d_2, ..., d_k\}$

b an initial direct environment $\mathbf{DE}_0$

c an empty set $\mathbf{LD}_0$

We define a partial subset of **LD**, $\mathbf{LD}_i$, as follows:

$$\mathbf{LD}_i = \{d_i\} \cup \mathbf{LD}_{i-1} = \{d_1, d_2, ..., d_i\} \tag{19}$$

and the direct environment associated with each declaration:

$$\mathbf{DE}_i = \mathbf{LD}_i \underset{m}{\cup} \mathbf{DE}_{i-1} \tag{20}$$

It is now a straightforward exercise to show that (18) follows (Appendix C Theorem 33). Note that for completeness, we must include the implicit declarations of named blocks as well as well as the explicit declarations from the declarative region in the set **LD**.

4.6 Visibility Within a Declaration

4.6.1 Let* Semantics

While discussing `let*` semantics in the previous section, we did not discuss the origin of $\mathbf{DE}_0$. The selection of this set is important, for it affects the visibility of a declaration (a function, for example) within itself. We must first extend our notation to indicate which declarative region that each set of declarations is associated with. We shall use a subscript to indicate which specific direct environment belonging to that declarative block we are referring to. For example, **X**:$\mathbf{DE}_Y$ would be the direct environment belonging to declarative region **X** that is associated with the declaration of **Y** in module **X**.

Consider the program given in example 2-5. If we let **factorial**:$\mathbf{DE}_0$ = **X**:$\mathbf{DE}_b$, where **X**:$\mathbf{DE}_b$ represents the incremental direct environment in **X** *just after the declaration of* `b`, then `factorial` is *not* visible within itself. On the other hand, if we let **factorial**:$\mathbf{DE}_0$ = **X**:$\mathbf{DE}_{factorial}$ then `factorial` *is* visible within itself.

```
X: declare
      b : integer;
      function factorial(a:integer) return integer is
      begin
         if a = 0 then
            return 1
         else
            return a * factorial(a-1);
         end if;
      end factorial;
```

```
   begin
      b := factorial(17);
   end X;
```

Example 10 Recursive Reference Nested Within a Declaration

Another situation that warrants consideration is depicted in Example 11. In the declaration of a that occurs within the block **Y**, there is also a *reference* to a that is part of the initialization. In which direct environment do we attempt to locate the declaration referred to by the reference to a? Two obvious choices are analogous to those that we used for $\mathbf{DE}_0$ above: $\mathbf{X}{:}\mathbf{DE}_a$ or $\mathbf{X}{:}\mathbf{DE}_Y$. Athird possibility, and the one used ***in Ada***, dictates that ***within the declaration, the name of the item being declared is not allowed to be used at all*****.** **T**o model this, we introduce a new environment $\mathbf{X}{:}\mathbf{DE}'_Y$ for use within the declaration that is identical to $\mathbf{X}{:}\mathbf{DE}_Y$ save that it does not contain any declarations with the identifier "a":

$$\mathbf{X}{:}\mathbf{DE}'_Y = \{d \in \mathbf{X}{:}\mathbf{DE}_Y \mid \text{identifier}(d) \neq \text{“a”}\} \tag{21}$$

It is in this environment that we seek to resolve references that occur within the declaration. Note that this alternative is also a third possible candidate for $\mathbf{DE}_0$.

```
X: declare
      a : integer := 4;
   begin
Y:    declare
         a : integer := a; -- illegal Ada!
                           -- where does the initial value
                           -- come from?
      begin
```

```
        null;
    end;
end;
```

Example 11 Recursive Reference Not Nested Within a Declaration

4.6.2 Compilation Diagnostic Aids

The approach used in (21) has the disadvantage that if a reference fails to locate the desired declaration in $\mathbf{X}$:$\mathbf{DE}'_Y$ it is impossible to determine just by looking at $\mathbf{X}$:$\mathbf{DE}'_Y$ whether a declaration was not found because there was no visible declaration or because the otherwise visible declaration was filtered out because of a name conflict. An alternative approach is:

$$\mathbf{X}:\mathbf{DE}'_Y = \{(a,\mathbf{0})\} \underset{m}{\cup} \mathbf{X}:\mathbf{DE}_Y \tag{22}$$

where (a,**0**) is a "fake" declaration that is a homograph of anything named "a". In this way, references that resolve to a dummy declaration will indicate name masking, while references that find nothing will indicate missing declarations.

4.6.3 Letrec semantics - Fully Recursive Referencing

In the `letrec` semantics of Scheme, the direct environment of the block containing the declaration serves as the source for all references that occur within the block, regardless of location. In Example 11, since the reference to a occurs immediately within the body of $\mathbf{Y}$, we look in $\mathbf{Y}$:**DE** to locate the declaration that is being referred to, which in this case is (a,$\mathbf{a} \in \mathbf{Y}$). We thus have a recursive reference, which, in this particular case, is undesirable. Note that this style of visibility computation places absolutely no restrictions on references that occur within declarations.

This semantics implies that machines are visible to the references that define the machines, i.e. the machines may be recursively defined. For example, since (Y,**Y**)∈**X**:**LD**, then (Y,**Y**)∈**X**:**DE** (by (16)), and (Y,**Y**)∈**Y**:**DE** (also by (16)) unless **Y**:**LD** contains a homograph of (Y,**Y**).

4.6.4 Let semantics - Non-recursive referencing

In let semantics, the new declaration of a is not considered to exist until the end of the declarative region in which a occurs. Thus, in example 2-6, we would look in **X**:**DE** to locate the referenced declaration, and thus locate (a,**a**∈**X**). This semantics also implies that declarations are not visible to references nested within themselves.

4.7 Ada Declarative Region Visibility (partial)

The visibility rules of Ada [LRM] utilize the name restricted referencing of (22) within declarations that do not, in turn, contain other declarations. For declarations that do contain other declarations, there is some variation in the computation of $\mathbf{DE}_0$. For a package specification, generic package specification or body **Y**, $\mathbf{Y}{:}\mathbf{DE}_0 = \mathbf{X}{:}\mathbf{DE}_Y$, where **Y** is a local declaration of **X**. For any other specification Y, $\mathbf{Y}{:}\mathbf{DE}_0 = (a,\mathbf{0}) \overset{\cup}{m} \mathbf{X}{:}\mathbf{DE}_Y$. Note that the declarative region containing formal parameter specifications is actually part of a specification, even though it might be contained in a subprogram body declaration, and all references in this region must use the name restricted referencing of (22).

4.8 Referencing Declarations from Other Scopes

In the previous sections, we have associated a distinguished pair of environments, **LD** and **DE**, with each block. We shall now use these environments to describe Ada's selected

component or “dot” notation. This notation is intended to make it possible to reference a declaration that might not otherwise be visible or unambiguous at a particular point in the program.

The definition of the ***dot notation*** in our model is as follows: Given the expression `X.Y` occurring in context **Z**, if there is a unique (X,**X**) in **Z**’s direct environment, and a unique (Y,**Y**) in **X**’s local declaration set, then **Y** is the designated declaration. More formally,

> if ∃(X,**X**) ∈ **Z**:**DE** and **X** is uniquely designated (there are no other declarations with the identifier X), and
>
> ∃(Y,**Y**) ∈ **X**:**LD** and **Y** is uniquely designated
>
> then **Y** is the designated declaration.

To illustrate this, consider the program of Example 12, in which we find the following sets:

> **X**:**LD** = {(a,**a**∈**X**), (a,**Y**)}
>
> **X**:**DE** = {(a,**a**∈**X**), (a,**X**), (a,**Y**), (integer,**Int**)}
>
> **Y**:**LD** = {(a,**a**∈**Y**)}
>
> **Y**:**DE** = {(a,**a**∈**Y**), (a,**X**), (a,**Y**), (integer,**Int**)}

Note that these sets have the same contents regardless of which declaration semantics is used.

```
X: declare
      a : integer;
   begin
```

```
Y:    declare
         a : integer;
      begin
         X.a := 1;
         Y.a := 2;
         X.Y.a := 3;
         X.X.a := 4; -- error: fails to resolve a
      end Y;
   end X;
```

Example 12 Referencing Declarations from Other Scopes

In the first assignment statement, we have the expression `X.a`, with semantics "find **X** in **Y**:**DE**." **X** must designate a block (we will generalize this later), and we look for a in **X**:**LD**, thus finding **a**∈**X**. In the second assignment statement, we look for **Y** in **Y**:**DE**, and then look for a in **Y**:**LD**, thus finding **a**∈**Y**. In the third statement, we first find **X** in **Y**:**DE**, then find **Y** in **X**:**LD**, and finally find a in **Y**:**LD**, thus locating **a**∈**Y**. In the fourth and final statement, we have a computation that fails: we can resolve the first reference to **X** in **Y**:**DE**, but there is no **X** in **X**:**LD**. Thus the reference in this statement is illegal.

While this explanation of dot notation is intuitively simple, it must be extended in order to represent Ada's use of the construct. Consider the program fragment in Example 13.

```
X: declare
      a : integer;
      package Y is
         a : integer :=1;
         b : integer := X.a;
         c : integer := X.Y.a;
```

```
      d : integer := X.Z.a; -- error: fails to resolve in Ada
   end Y;
   package Z is
      a : integer;
      b : integer := X.a;
      c : integer := X.Y.a;
      d : integer := X.Z.a;
   end Z;
begin
   null;
end X;
```

Example 13 Declaration Ordering Affects Dot References in Ada

Here we have used Ada packages, which, for the purpose of this example, can be considered to be blocks whose declared machines continue to exist beyond the execution of the code within the blocks. In this example, **X**:**LD** contains both **Y** and **Z**, but the reference to **Z** within **Y** is illegal in Ada, since **Z** was not declared (sequentially) until after the declaration of **Y**. (Note also that if **Y** and **Z** had been blocks, the references to **X**'s declarations would not have been legal, since they do not persist beyond the scope of the block.) Our present interpretation of the dot notation would successfully locate **Z**, and is thus inconsistent with the Ada interpretation. Rather than looking in **X**:**LD** for **Z**, we should have looked in $\mathbf{X}{:}\mathbf{LD}_{Y}$, which is the incremental local declaration set that existed immediately after the declaration of **Y**.

This solution must be generalized to deal with a lexical ancestor, no matter how far removed, anywhere within the dot notation. We begin by defining a new relation, ***P***(**A**, **B**), the parent relation, where **A** and **B** are both machines, and $\mathbf{B} \in \mathbf{A}$. We next define the or-

dered set of ancestors of **B** as *Ancestors*(**A**) = $\{d_1, d_2, \ldots d_k \mid d_k = B$ and $\forall i\ \boldsymbol{P}(d_i, d_{i+1})\}$. We now consider a ***first approximation to Ada dot notation***:Given the expression `X.Y` occurring in context **Z**, if there is a unique **X** (either found as (`X`,**X**) in **Z**'s direct environment if `X` is the first element in a dot expression, or located by a previous evaluation of the portion of the dot expression to the left of the dot being currently considered), then there are four cases:

1 Case 1: **X**=**Z** (**X** is the current context): if there is a unique (`Y`,**Y**) in the current **X**:**LD**, then **Y** is the designated machine.

2 Case 2: **X** is an ancestor of **Z** (**X** $\in$ *Ancestors*(**Z**)): let i be the position of **X** in *Ancestors*(**Z**). If there is a unique (`Y`,**Y**) $\in$ $\mathbf{X}\text{:}\mathbf{LD}_{d_{i+1}}$, then **Y** is the designated machine, where d_{i+1} is the immediate descendant of **X** that is also an ancestor of **Z**.

3 Case 3: **X** is not an ancestor of **Z** (**X** $\notin$ *Ancestors*(**Z**)) and **X** is a parallel machine with respect to the current context: if $\exists$(`Y`,**Y**) $\in$ **X**:**LD** and **Y** is uniquely designated (it is the only machine associated with `Y` in this environment) then **Y** is the designated declaration. Typical examples of parallel machines are variables, packages and task instances. An example of a non-parallel machine is a subprogram: its component elements do not exist until the subprogram is called, and do not persist past the end of the subprogram's execution. The only time that these elements may be referenced is during the execution of the subprogram, and this is covered in case 1 or 2.

4 Case 4: **X** is not an ancestor of **Z** (**X** $\notin$ *Ancestors*(**Z**)) and **X** is not a parallel machine: then none of the declarations of **X** are parallel, and the reference to **Y** is meaningless.

For an expression involving multiple dots, such as `A.B.C`, the expression is evaluated left-to-right, using the algorithm above: first `A.B` is resolved, then the resulting **B** is used in resolving `B.C`.

5.0 Applying Constructive Semantics

The intent of constructive semantics is to provide an interpretation of a program as the specification for an abstract state machine. We are now in a position to specify the constructive semantics of a programming language using the results of the earlier chapters. The style of this semantics will be denotational, showing how each construct in the language can be interpreted as a machine that is defined by the composition of the denotations of its component parts.

A programming language defines a number of basic data types and operations as given elements of the language. We assume that the machines that these data types and operations denote are given as part of the formal semantics of the language.

We will take a subset of Ada as the basis for showing how a constructive semantics for a programming language can be given. This subset includes many of the language features of Ada, including declarative blocks, composite types and exception handling. Packages and tasks have been omitted because they differ little from declarative blocks in their semantics except for their extremely complex visibility computation rules, which are analyzed and modeled in [Brow90].

In giving the semantics for each construct, we will not provide a complete Ada semantic, but rather explore a variety of semantics that might be given to the construct. In most cases, one of the semantics that we will give closely approximates the actual Ada semantics.

5.1 Elaboration: From Programs to Machines

If a program is the specification of a state machine, somewhere along the line this specification must be converted into an actual machine. While one might be initially tempted to say that this is the role of the compiler, the compiler in reality simply generates a series of

binary numbers in a file that represent the *initial state* of some other machine, namely the computer in which the program will be executed.

Converting a specification into a state machine is not necessarily a one step process. The obvious counter-example is the interpreter, which does the conversion piecemeal. But even a compiled language may not be convertible into a complete state machine at load time. For example, some Ada data type declarations (which we shall be interpreting as definitions of machine types) are allowed to depend upon values computed previously in the program. Thus the machines defined by these type declarations are not even fully defined until part of the program has been executed.

In providing a semantics for a program, we are not concerned about the details of *how* a compiler converts a program into a state machine. Our concern is to specify the *behavior* of that machine after conversion. However, we shall have frequent occasion to refer to this process of conversion, and thus, borrowing a term from Ada, we shall call this process ***elaboration***. In fact, the method of specifying our semantics is simply to give a formal definition of elaboration.

5.2 Concepts of Type

The term type frequently brings to mind differing and possibly inconsistent concepts. The three dominant concepts seem to be that a type is either a set of values[1], a behavioral (interface) specification, or an implementation specification. Our semantics provides a perspective from which we see that these three concepts are not inconsistent, and are in fact closely related to one another in a very formal way.

1. Or a representative of the set, as is the case in Denotational Semantics

One simplifying factor resulting from the state machine perspective is that values, per se, *do not exist* in the state machine model! The closest that we can come to the concept of a value belonging to a set of values is the current state of a value machine capable of encoding the values from the set. But there is an important aspect of the value concept that must be considered: the notion of equivalence of values. This concept is provided by the state type definition of section 2.5. If we assume that the machines of the state type are value machines, then the isomorphic relationship between the machines (actually between the actions of one machine and the actions of another) enables us to establish the equivalence of values being represented by different machines (we will later discuss a machine that actually tests membership in this relation).

Now when we are working with (interacting with) a value machine, we are not directly observing the state of the machine (i.e. the encoding of the value being represented): we are only able to ***infer*** the intended value from the actions to which the value machine will respond. For example, if the value machine shown in Figure 11 happens to be in state V_1, then it will interact via action `o1`, indicating that the value 1 was being represented, but would not interact via the action `o2` or `o3` which would indicate that some other value was being represented.

What we learn from this is that the only thing that we can observe about a machine is its behavior in terms of its interactions with other machines (its actions). Our first notion of type has thus become a special case of the second notion, i.e. a behavioral specification. In most cases, we shall see that ***the declaration of a type in a programming language is both a behavioral specification and the specification of a machine that implements that behavior***.

5.2.1 Simple Data Types

The simplest notion of type that we find in programming languages is that of a set of values, and these values are distinct from the values of every other type. For each set of values that we wish to represent, we define a state type of value machines capable of representing that set of values. The pre-defined or built-in types of a language are readily modeled as pre-declarations of a number of state types.

5.2.2 Types with Structure

In modern programming languages, we have begun to see more complex data types arise: the C union, the Ada variant record, and the subclass in C++ to name just a few. While we will not give semantics for these types in this work, it is worth outlining the general approach to modeling these types in constructive semantics.

The C union is a data structure that may take more than one form. If $\mathbf{M}_1,\mathbf{M}_2,...,\mathbf{M}_n$ are the state types for the various alternatives, then a first attempt at a model might take the form:

$$\mathbf{M}_1 \oplus \mathbf{M}_2 \oplus ... \oplus \mathbf{M}_n$$

This would partially capture the desired semantic. Unfortunately, if this were to be used as the model for a variable, once an action belonging to one of the machines has occurred, then the machine could never take on values belonging to the other types. To model this situation, we need to add an additional machine, similar in structure to the pointer machine that we will describe later, that will keep track of which machine holds the “current” value, only allowing read operations on that machine but allowing write operations on all of the machines. Such a composition might take the general form:

$$\mathbf{C} \mid \mathbf{M}_1 \mid \mathbf{M}_2 \mid ... \mid \mathbf{M}_n$$

where **C** is now the controlling machine, and the actions of $\mathbf{M}_1 | \mathbf{M}_2 | ... | \mathbf{M}_n$ would be hidden (**C** acting as an intermediary for all operations).

In variant records, the machine **C** becomes explicit - it is the variant part of the record. From an external point of view, all variants of the record share the behavior of **C**, and depending upon the value that **C** currently holds, the machine may behave like $\mathbf{C} | \mathbf{M}_1$ or $\mathbf{C} | \mathbf{M}_2$ or any of the possible variants of the record. Stated another way, **C** defines an abstract type (recall that using an abstract type says that the actions are restricted to those of **C** for the purposes of determining behavioral equivalence) that is common to all of these machines. Thus the state types defined by $\mathbf{C} | \mathbf{M}_1$ and $\mathbf{C} | \mathbf{M}_2$ and the rest of the possibilities are all of the same abstract type **C**. This readily generalizes into a model for object oriented languages, in which **C** is the parent class, and $\mathbf{C} | \mathbf{M}_1$, $\mathbf{C} | \mathbf{M}_2$ etc. would be its children. We also note that, since a state type can be of more than one abstract type, the model covers multiple inheritance as well.

5.3 Supporting Data Structures and Functions

We define some auxiliary data structures and operations for use in giving the actual semantics of expressions.

5.3.1 Environments

An ***environment*** $E : Ide \times M$ is a relation between identifiers and machines.

5.3.2 State Types

The approach that we shall take to defining state types is to give the specification of the prototype machine and then keep track of the machines of that type as they are created. To record these relationships, we define the relation *StateT*ype:

$$StateType: M \times M$$

where the first machine is the machine whose type is being given, and the second machine is the prototype machine of the state type. Note that the prototype machine is in the relation mapped to itself.

5.3.3 Abstract Types

We keep track of the relationships between types by mapping the prototype machine of the subclass to the prototype machine of its parent type in the *AbstactType* relation:

$$AbstractType: M \times M$$

Where both machines are prototype machines, with the first type being a subclass of the second.

5.3.4 Signatures of Subprograms

In order to do overload resolution, we must keep track of the parameter and result type profile of subprograms. We define the *Signature* relation for this purpose, where:

$$Signature: M \times M^*$$

maps functions (the state type defining the function) onto an ordered list of the state types associated with their arguments and return types. In this relation each entry is of the form

($\mathbf{M}$,{$\mathbf{M}_0$,$\mathbf{M}_1$,$\mathbf{M}_2$,...,$\mathbf{M}_n$}) and {$\mathbf{M}_0$,$\mathbf{M}_1$,$\mathbf{M}_2$,...,$\mathbf{M}_n$}$\in M^*$ is an ordered list of state types. We note that if the state type on the left is not a subprogram, then *Signature* maps the state type to itself. We further note that each state type only appears once on the left side of this relation. We establish the convention that the first element of the signature is the return type, and we shall use the idle machine **0** for this first element if the subprogram is a procedure.

5.3.5 Type Structure

We gather these three elements together into a data structure that we shall call a ***type structure***. We define a type structure T to be a triple T=(*StateType*,*AbstractType*,*Signature*)

5.3.6 Finding the Type of a Machine

We define the semantic function **Type**:$M \rightarrow M$ that returns the type of a machine. Formally, this function is defined by:

$$\mathbf{Type(M) = M_T} \text{ iff } (\mathbf{M}, \mathbf{M_T}) \in StateType$$

5.3.7 Finding the Argument Signature of a Machine

We define the semantic function **ArgSig**:$M \rightarrow M^*$ that returns the signature of the arguments of a machine. Formally, this function is defined by:

$$\mathbf{ArgSig(M) = M_1,M_2,...,M_n} \text{ iff } \exists \mathbf{M_0} \text{ s.t. } (\mathbf{M}, \{\mathbf{M_0},\mathbf{M_1},\mathbf{M_2},...,\mathbf{M_n}\}) \in Signature$$

where {$\mathbf{M}_0$,$\mathbf{M}_1$,$\mathbf{M}_2$,...,$\mathbf{M}_n$} is an ordered set of state types. A consequence of the way that *Signature* is defined is that if $\mathbf{M}$ is not a subprogram, then **ArgSig(M)** = {}.

5.3.8 Finding the Return Signature of a Machine

We define the semantic function **RetSig**:$M \rightarrow M$ that yields the return type of a machine. Formally, this function is defined by:

$$\textbf{RetSig(M)} = \mathbf{M}_0 \quad \text{iff } \exists \mathbf{M}_1,\mathbf{M}_2,...,\mathbf{M}_n \text{ s.t. } (\mathbf{M}, (\mathbf{M}_0,\mathbf{M}_1,\mathbf{M}_2,...,\mathbf{M}_n)) \in \textit{Signature}$$

A consequence of the way that *Signature* is defined that if **M** is not a subprogram, then **RetSig(M)** = **M**

5.3.9 Finding the Signature of a Machine

We define the semantic function **Sig**:$M \rightarrow M^*$ that returns the signature of a machine type. Formally, this function is defined by:

$$\textbf{Sig(M)} = \mathbf{M^*} \text{ iff } (\mathbf{M},\mathbf{M^*}) \in \textit{Signature}$$

We note that for all types except those representing functions,

5.3.10 References

We present a slightly simplified view of references and overload resolution here, postponing the formalization of more elaborate overload resolution schemes for later work. When we seek to locate a machine by name in an environment, we will either not use any type information, or we will specify explicitly the argument signature for the subprogram that we are seeking.

We define the simple reference operation **Ref**:$Ide \times E \rightarrow M$ to be a mapping from environments and identifiers to machines. Formally, this operation is characterized by:

$$\textbf{Ref(}\mathit{ide},\textbf{ENV)} = \mathbf{M} \text{ iff } (\mathit{ide},\mathbf{M}) \in \textbf{ENV} \text{ and } \forall \mathbf{M'} \neq \mathbf{M}, (\mathit{ide},\mathbf{M'}) \notin \textbf{ENV}$$

Note the requirement that the machine be uniquely identified. If no machine is uniquely identified, we say that the reference is undefined, and we require that all of our semantic functions be strict in the sense that any composition containing an undefined machine is, itself, undefined.

We define the typed reference operation $\mathbf{TypRef}: Ide \times E \rightarrow M$ to be a mapping from environments, signatures and identifiers to machines. Formally, this operation is characterized by:

$$\mathbf{TypRef}(ide, \mathbf{ENV}, \text{sig}) = \mathtt{M} \text{ iff } (ide, \mathtt{M}) \in \mathbf{ENV} \text{ and } \mathbf{ArgSig}(\mathtt{M}) = \text{sig and}$$

$$\forall \mathtt{M}' \neq \mathtt{M}, (ide, \mathtt{M}') \in \mathbf{ENV} \Rightarrow \mathbf{ArgSig}(\mathtt{M}') \neq \text{sig}$$

Note again the requirement that the machine be uniquely identified.

5.3.11 Getting New Machines of a Type

In our semantics we will frequently have a prototype machine (a state type) and wish to get a new machine of that type. We define the semantic function $\mathbf{New}: M \rightarrow M$ for this purpose. What this function really does is to choose an index i into the machines of this type that has not been used yet and relabel the prototype machine according to this index:

$$\mathbf{New}(\mathtt{M}) = \mathtt{M}[\phi_{0 \rightarrow i}]$$

5.4 Elaboration: Formal Definition

Elaboration is a mapping from a language term, a local declaration set, a direct environment and a type structure to a machine:

$$\mathbf{E}: L \times E \times E \times T \rightarrow M$$

The first term is the language expression whose meaning is being determined. The second term is an environment that is to contain local declarations. Typically, this environment is modified in the course of elaboration. The third term is an environment that contains machines that have been defined elsewhere that may be used in the course of performing the elaboration.The fourth term is a type structure, which may contain some information initially and may also be added to during the course of elaboration.

5.5 Basic Data Types and Operations

The state types for primitive data types and their associated operations are taken as being given as part of the language specification. In a formalization of the language, specifications of these machines in HCCS should be given so that there is no ambiguity about the behavior of these machines. We have already given an example of a primitive data type (the 3-valued variable in section section 3.1.2 on page 50) and an associated operation (the assignment operator for 3-valued variables in section 3.2 on page 54).

5.6 Variables

A variable is a machine for holding values. The type of the machine is either partially or entirely specified by the type of the variable, depending upon whether or not the abstract type declared by the variable has other state types (subtypes) related to it that may appear as actual values. For the purpose of discussion, we shall consider a state type that does *not* have any subtypes to be a ***simple type***, and a state type that *does* have subtypes to be a ***complex type***. We note that the subtype relation in the type structure contains the information necessary to decide whether a type is simple or complex.

We will describe the semantics of a variable of a simple data type here, and leave the semantics of complex types for a later work.

5.6.1 Variables of a Simple Type

Variables of a simple data type are modeled as a value machine belonging to the indicated state type. We locate the prototype machine of the state type and make a new copy:

$$\mathbf{E(}[\![\ \texttt{variablename: typename;}\ ]\!]\mathbf{,\ LD,\ DE,}\ T\mathbf{)}$$

$$= \quad \mathbf{M} \equiv \mathbf{New(M_T)}$$

where

$$\mathbf{M_T} = \mathbf{Ref(}\texttt{ide}\mathbf{,DE)}$$

with side effects adding the new declaration to the set of local declarations and recording the type relationship in the *StateT*ype relation:

$$\mathbf{LD} = \mathbf{LD} \cup \{(\texttt{variablename}, \mathbf{M})\}$$

$$\textit{StateT}\text{ype} = \textit{StateT}\text{ype} \cup (\mathbf{M},\mathbf{M_T})$$

5.7 Pointers

Pointers are an indirect means of accessing a machine of a particular type. We give a semantics here for a strongly typed pointer. The machine we will use to represent the pointer will actually have two components, one being a variable that indicates which machine is being pointed to, and the other being an interaction machine that re-directs actions to the appropriate machine.

In order to define the variable, we must first define its state type. Let $\boldsymbol{T}$ be the abstract type that we wish to have a pointer to, and let $\textit{AbstractIndex}(\mathbf{M}, \boldsymbol{T})$ be a mapping from machine $\mathbf{M}$ to its *index* in the set of machines belonging to the specified abstract type $\boldsymbol{T}$. For every

abstract type $\mathbf{T}$ we assume that there is a corresponding state type ***TAbstractIndex*** capable of representing the values of the indices. Then the variable part of the pointer is just a variable of this type:

$$\mathbf{V}_P \equiv \mathbf{New}(\mathit{TAbstractIndex})$$

We let $o1_V$ be the action that the variable would take to indicate that the pointer is pointing to the first machine belonging to abstract type $\mathbf{T}$, $o2_V$ indicate that the second machine is being pointed to, and so on.

Now we define the interaction machine. Our strategy is to use the actions belonging to one member of the abstract type (we will use the actions belonging to the machine whose index is 0) as the *interface* to the access type, and re-direct these actions to the machine designated by the variable. Let $a_{i,j} \in Act_\omega$ be the jth action on the ith machine belonging to the abstract type. Then we define a prototype interactor $\mathbf{I}_P$ for all possible machines of abstract type $\mathbf{T}$ with the following equations, letting I be the initial state:

$$I \equiv \alpha.I_0$$

$$\begin{aligned} I_0 \equiv\ & a_{0,0} o1_V a_{1,0}.I_0 \oplus a_{0,1} o1_V a_{1,1}.I_0 \oplus \ldots \oplus a_{0,n} o1_V a_{1,n}.I_0 \oplus \\ & a_{0,0} o2_V a_{2,0}.I_0 \oplus a_{0,1} o2_V a_{2,1}.I_0 \oplus \ldots \oplus a_{0,n} o2_V a_{2,n}.I_0 \oplus \\ & \ldots \\ & a_{0,0} om_V a_{m,0}.I_0 \oplus a_{0,1} om_V a_{m,1}.I_0 \oplus \ldots \oplus a_{0,n} om_V a_{m,n}.I_0 \oplus \\ & \iota.I \end{aligned}$$

where n is the number of actions on a machine of this abstract type, and m is the number of machines. Each term corresponds to a single action on a single target machine. For exam-

ple, the action $a_{0,i} \circ m_V a_{m,i}$ occurs if and only if another machine attempts to interface with action $a_{0,i}$, the pointer is currently pointing to machine m (indicated by the variable responding to $\circ m_V$) and the machine being pointed to responds to the action $a_{m,i}$.

Thus the semantics of the access type declaration in Ada would be:

$$\mathbf{E}(\llbracket \texttt{ type } \textit{accessname} \texttt{ is access } \textit{typename}\texttt{;} \rrbracket, \mathbf{LD}, \mathbf{DE}, T)$$
$$= \mathbf{M} \equiv \langle \mathbf{V}_P \mid \mathbf{I}_P \rangle$$

where $\mathbf{V}_P$ and $\mathbf{I}_P$ are as defined above and the abstract type $\boldsymbol{T}$ = **Ref(***typename*,**DE)**. We also have the following side effects:

$$StateType = StateType \cup (\mathbf{M},\mathbf{M})$$

$$AbtractType = AbtractType \cup (\mathbf{M},\boldsymbol{T})$$

$$AbtractType = AbtractType \cup (\mathbf{M},\boldsymbol{TAbstractIndex})$$

Note that we have a case here in which a single state type belongs to two abstract types. When restricted to the actions of ***TAbstractIndex***, it behaves like a variable of that type (so that we can set the value of the pointer). When restricted to the actions of ***T*** it behaves like a machine of that type so that we can interact with machines of type ***T***. We thus have an instance of what is commonly referred to as ***multiple inheritance***.

5.8 Statements

Statements are the basic unit of execution in a programming language. The most basic statement is the direct invocation of an operation such as in an assignment statement or the invocation of a subprogram. In covering the semantics of subprograms, we shall see that

the semantics of operators and subprograms is uniform both at the statement level and within expressions.

Next, we shall proceed to give semantics for the infamous goto statement, followed by its modern counterpart, raining an exception. Finally, we will conclude the discussion of statements with the semantics of a loop.

5.8.1 Subprogram Calls and Expression Evaluation

The semantics that we give for subprogram calls encompasses the invocation for both operators and subprograms. This treatment of operators as pre-defined subprograms allows user-defined versions of these operators to be declared in a program and used consistently throughout the program. The newly defined operator affects the visibility of the original operator according to the visibility rules of the language

We will give several representative formulations for calling two argument subprograms which can be readily generalized to an arbitrary number of arguments. In defining these expressions, we will frequently need to know the name of the operation at the root of an expression. We define the syntactic function *Root*(expression) that returns this identifier.

The simplest form of subprogram call assumes that the arguments are simply references to existing machines (thus requiring no elaboration of the arguments) and further assumes that the arguments are of the correct type. We arrive at a relatively simple semantic for our subprogram call:

$$\mathbf{E}(\llbracket \texttt{subprogramName (<argument}_1\texttt{>, <argument}_2\texttt{>);} \rrbracket, \mathbf{LD}, \mathbf{DE}, T)$$

$$= \mathbf{New}(\mathbf{Ref}(\texttt{subprogramName}, \mathbf{DE}))[\mathbf{M}_1/\mathbf{P}_1, \mathbf{M}_2/\mathbf{P}_2] \rangle$$

where $\mathbf{M}_1$ = **Ref(***Root*(<argument$_1$>)**)**, $\mathbf{M}_2$ = **Ref(***Root*(<argument$_2$>)**)**, and $[\mathbf{M}_1/\mathbf{P}_1,\mathbf{M}_2/\mathbf{P}_2]$ is a minor abuse of our mapping notation indicating that the actions of the formal parameter $\mathbf{P}_1$ are mapped to the actions of actual parameter $\mathbf{M}_1$, and similarly for $\mathbf{P}_2$ and $\mathbf{M}_2$. This assumes, of course, that $\mathbf{P}_1$ and $\mathbf{M}_1$ are of the same type, and similarly $\mathbf{P}_2$ and $\mathbf{M}_2$ are of the same type.

We now extend the semantics to include a modest amount of type checking. In this scheme, type information propagates only in one direction: upwards from actual arguments to functions.

E(⟦subprogramName (<argument$_1$>, <argument$_2$>);⟧, **LD**, **DE**, *T***)**

= New(TypRef(subprogramName, **DE**,

{**RetSig(**$\mathbf{M}_1$**)**×**RetSig(**$\mathbf{M}_2$**)**}**))**$[\mathbf{M}_1/\mathbf{P}_1,\mathbf{M}_2/\mathbf{P}_2]$

The semantics given for the previous two examples will work correctly if the argument is an already instantiated machine instance as opposed to the prototype machine of a state type. If the reference to the root of an argument turns up a prototype machine instead of an instance, then we must create a new instance of that state type by elaborating the argument as part of the elaboration of the subprogram call. We thus get (ignoring type checking again):

E(⟦subprogramName (<argument$_1$>, <argument$_2$>);⟧, **LD**, **DE**, *T***)**

= ⟨ **New(Ref(**subprogramName, **DE))**$[\mathbf{M}_1/\mathbf{P}_1,\mathbf{M}_2/\mathbf{P}_2]$ |

if **Type(**$\mathbf{M}_1$**)** = $\mathbf{M}_1$ then **E(**⟦<argument$_1$>⟧, **LD**, **DE**, *T***)** else **0** |

if **Type(**$\mathbf{M}_2$**)** = $\mathbf{M}_2$ then **E(**⟦<argument$_2$>⟧, **LD**, **DE**, *T***)** else **0** ⟩

This almost gives us the semantics that we want, except for a possible problem in the order of evaluation: we have not constrained the arguments to ensure that they are evaluated before the subprogram itself is evaluated. To accomplish this, for each argument that requires elaboration, we introduce a new local variable to carry the result of the argument evaluation forward to the subprogram itself. Ignoring type checking again, and leaving out the conditionals (we assume that both arguments require elaboration) we now have:

$$\mathbf{E(}[\![\ \textit{subprogramName}\ \texttt{(<argument}_1\texttt{>, <argument}_2\texttt{>);}\ ]\!]\mathbf{,\ LD,\ DE,}\ T\mathbf{)}$$

$$\begin{aligned} = \ \langle\ \mathbf{V_1} \mid \mathbf{V_2} \mid \quad & \mathbf{E(}[\![\texttt{<argument}_1\texttt{>}]\!]\mathbf{,\ LD,\ DE,}\ T\mathbf{)}[\mathbf{V_1}/\text{out}]; \\ & \mathbf{E(}[\![\texttt{<argument}_2\texttt{>}]\!]\mathbf{,\ LD,\ DE,}\ T\mathbf{)}[\mathbf{V_2}/\text{out}]; \\ & \mathbf{New(Ref(}\textit{subprogramName}\mathbf{,\ DE))}[\mathbf{V_1}/\mathbf{P}_1,\mathbf{V_2}/\mathbf{P}_2]\ \rangle \backslash \textit{Sort}(\mathbf{V_1}) \cup \textit{Sort}(\mathbf{V_2}) \end{aligned}$$

where $[\mathbf{V_1}/\text{out}]$ relabels the output formal parameter with the actions of $\mathbf{V_1}$. Note that we have somewhat arbitrarily determined an order of evaluation for the actual parameter expressions.

In [Brow89] we have explored event more complex type checking and overload resolution schemes. While these schemes add significantly to the type information that is passed around between references (the reference functions themselves become very complicated), the basic structure of the elaboration in terms of the structure of instantiated machines still remains the same.

A final note on subprogram calls. We have made no distinction between functions and procedures in this semantics, nor have we made any distinction between calls that occur as an actual statement and calls that occur as part of a subexpression. We note that if one of the actual arguments to the subprogram was accidentally a procedure, then the typed reference

to the subprogram would fail to resolve to a machine. This illustrates the generality of this approach to semantics.

5.8.2 Go-To

One of the oldest control constructs is the go-to statement. While a go-to appears to be a simple jump to the start of a particular machine's operation (its activation), we will almost certainly be in the middle of executing another machine when we execute the go-to. If we wish to exit this machine in such a way as to leave it ready to execute again, then we must add an ***abort action*** ♦ to return the machine to its initial state without performing the idle action. The following construct illustrates the use of the abort action in a trivial machine:

$$M_0 \equiv \alpha.M_1 \oplus \blacklozenge.M_0$$

$$M_1 \equiv a.M_1 \oplus \iota.M_0 \oplus \blacklozenge.M_0$$

Note that if the abort action occurs while the machine is in its initial state, the machine simply returns to its initial state. On the other hand, if the machine is in some other state, like M_1, the abort action returns the machine to its initial state *without performing the idle action* ι, which would indicate that the machine had terminated in a normal manner, and could well trigger other activity. This is the essence of the abort action: it returns the machine to its initial state without indicating that the machine has terminated normally. We note that the abort action, like the activate and idle action, would have to be unique to each machine.

We must also consider the problem of aborting a collections of machines in a parallel construct. Consider the following:

$$\langle M_1 \mid M_2 \mid ... \mid M_n \rangle$$

In giving the expansion of this term into HCCS (section 3.3 on page 57), we might now add an additional term that ties the abortion of the overall construct to the abortion of the individual machines in the construct:

$$(\overline{\blacklozenge}_1\overline{\blacklozenge}_2 \ldots \overline{\blacklozenge}_n \blacklozenge_X .)^*$$

where X designates the actions unique to the parallel construct itself. A machine can now abort the entire parallel construct by interacting with the $\blacklozenge_X$ action.

We note that the ability to abort the construct depends upon the willingness of each machine in the construct to perform an abort. This behavior must be designed into the machines involved, and is a significant issue in the specification of the semantics for a programming language, particularly when parallel constructs such as Ada tasks are involved. We shall also see shortly that in many cases we will not wish to abort all of the machines in the construct, but selectively abort some of them.

We now return to a discussion of the go-to statement. The destination statement of a go-to might be of the syntactic form:

```
label : <statement>
```

We will elaborate this construct into two machines, one being the normal statement elaboration, and the other providing an alternate means of activating this statement. The result of elaborating this construct is as follows:

L ; **S**

where **S** is the machine resulting from the elaboration of the statement itself, and **L** is a label machine (giving us the alternate means of activating **S**) of the form:

$$L_0 \equiv \alpha\iota.L_0 \oplus g\iota.L_0$$

Here g is a new symbol that implements the actual goto action by forcing the ι action of the label machine, which is, in turn, tied to the activate action of **S** via the ";" interactor. Thus any time that the g action occurs, the idle action for the label machine occurs, this triggering (through the ";" operator) the activation of the following statement.

The semantics for the labeled statement is thus:

$$\mathbf{E}(\llbracket \texttt{label : <statement>} \rrbracket, \mathbf{LD}, \mathbf{DE}, T)$$

$$= \mathbf{L'} ; \mathbf{E}(\llbracket \texttt{<statement>} \rrbracket, \mathbf{LD}, \mathbf{DE}, T)$$

where

$$\mathbf{L'} \equiv \mathbf{New(L)}$$

where **New(L)** gives us a fresh copy of **L** with new actions (in particular, a new g action unique to this label).

Now in order to effect the go-to, we of course require another machine to interact with this label machine in order to start the statement again. Accordingly, we define such a machine and associate this machine with the label name in the environment as part of the elaboration of the label itself, thus giving us the side effect:

$$\mathbf{LD} = \mathbf{LD} \cup (\texttt{label}, \mathbf{New(GoTo)}[g'/\texttt{jump}, \blacklozenge_{SOS}/\blacklozenge_S])$$

where we again abuse the relabeling notation to indicate that the action `jump` in the **GoTo** machine is mapped to the g action in the newly created **L'**. Also, the action $\blacklozenge_S$ of the **GoTo** machine is mapped to the abort action of the enclosing sequence of statements.

The **GoTo** machine itself is simply:

$$G_0 \equiv \alpha.\blacklozenge_G \overline{\blacklozenge}_S.\text{jump}.G_0 \oplus \blacklozenge_G.G_0$$

where $\overline{\blacklozenge}_S$ aborts just the machines in the enclosing sequence of statements, and $\blacklozenge_G$ is a dummy action provided on this statement so that the $\overline{\blacklozenge}_S$ action will succeed in aborting all of the statements in the sequence, including this one. Note that this machine does not, in fact, abort: it will perform the jump action before returning to its initial state. Note that this machine never executes an idle action.

Finally, the goto statement itself has the semantics:

E(⟦`goto` *label*`;`⟧**, LD, DE,** *T***)**

= New(Ref(*label*`,`**DE))**

To illustrate the behavior of the following program fragment:

```
T: b := 2*a;
   goto b;
```

Letting L_T be the label machine associated with `T`, M_A be the machine resulting from the elaboration of the assignment statement, and M_G be the goto machine, the elaboration of this fragment would result in the following machine composition:

$$L_T ; M_A ; M_G \mid (\blacklozenge_S \overline{\blacklozenge}_A \overline{\blacklozenge}_G.)^*$$

where $(\blacklozenge_S \overline{\blacklozenge}_A \overline{\blacklozenge}_G.)^*$ is the term added by a sequence of statements elaboration that aborts each of the statements in the sequence. Expanding into HCCS, we have:

$$L_0 \equiv (\alpha_L \iota_L.L_0 \oplus g_T \iota_L.L_0) \mid (\bar{\iota}_L \bar{\alpha}_A.)^* \mid M_A \mid (\bar{\iota}_A \bar{\alpha}_G.)^* \mid G_0 \equiv (\alpha.\blacklozenge_G \bar{\blacklozenge}_S.g_T.G_0 \oplus \blacklozenge_G.G_0)$$

Let us follow the behavior through one cycle. When L_0 is first activated, it will perform the action $\alpha_L \iota_L$, which will interact through $\bar{\iota}_L \bar{\alpha}_A$ to activate machine M_A (the assignment statement). When the assignment statement is through, it will execute its idle action, which, through $\bar{\iota}_A \bar{\alpha}_G$, will activate the goto machine. The goto machine will perform the action $\blacklozenge_G \bar{\blacklozenge}_S$, which interacts with the parallel $(\blacklozenge_S \bar{\blacklozenge}_A \bar{\blacklozenge}_G.)^*$ term to idle M_A (note the use of "dummy" $\bar{\blacklozenge}_G$). This gets all of the machines in the sequence of statements back to their initial state except for the goto statement itself. The goto now performs the g_T action, which interacts with $g_T \iota_L$ on the label machine, which in turn via $\bar{\iota}_L \bar{\alpha}_A$ activates the assignment machine again, thus implementing the goto. Note that the aborting of the sequence of statements implies that machines that have not been activated yet (are still in their initial state) must be willing to perform an abort action, returning to their initial states.

We have not, as yet, explored all of the implications of the abort action, and in particular we have not explored the implications of abort actions with respect to the parallelization, change of scope and removal of brackets theorems. We leave this as a topic for future work.

5.8.3 Raising an Exception

An exception in Ada is the very similar to a go-to statement, except that the machine to be activated is not in the normal sequence of statements, but is itself a separate machine. When an exception is raised, the sequence of statements in the declarative block where the exception handler is found is aborted (thus aborting all subordinate constructs), and the machine corresponding to the exception handler is activated. This would give the following semantics for a raise statement:

$$\mathbf{E(}[\![\ \texttt{raise}\ \textit{exceptionName};\ ]\!]\mathbf{,\ LD,\ DE,}\ T\mathbf{)}$$

$$= \mathbf{New(Ref(}\textit{exceptionName}\mathbf{,\ DE))}$$

The exception handler itself is defined by:

$$\mathbf{E(}[\![\ \texttt{when}\ \textit{exceptionName}\ \texttt{=>}\ \texttt{<sequence of statements>}\ ]\!]\mathbf{,\ LD,\ DE,}\ T\mathbf{)}$$

$$= \mathbf{L'}\ ;\ \mathbf{E(}[\![\texttt{<sequence of statement>}]\!]\mathbf{,\ LD,\ DE,}\ T\mathbf{)}$$

where $\mathbf{L'}$ is defined as in the go-to statement, and the appropriate goto machine is inserted in the local declaration set:

$$\mathbf{LD} = \mathbf{LD} \cup (\textit{exceptionName}, \mathbf{New(GoTo)}[g'/j, \blacklozenge_{SOS}/\blacklozenge_{S}])$$

where $\blacklozenge_{SOS}$ is the abort action associated with the normal sequence of statements in the declarative block, and not the sequence of statements provided here as part of the exception handler.

5.8.4 Loop

We give two alternative semantics for loops. The first semantic involves the instantiation of a generic loop machine with the conditional expression and sequence of statements as parameters. This will result in a construct in which each machine is activated and idled only once (if at all), but has the disadvantage that it is a recursive construct in which copies of the condition machine and the sequence of statements appear an infinite number of times. The generic loop machine $\mathbf{LOOP(C,S)}$ is given by:

$$\text{LOOP} \equiv \alpha.(\mathbf{V} \mid \text{LC})\backslash \mathit{Sort}(\mathbf{V})$$

$$LC \equiv (\mathbf{S} \mid \mathbf{C} \mid =^{\mathbf{C}\rightarrow\mathbf{V}} \mid \overline{\alpha}_C.\overline{\iota}_C.(\texttt{true}_V\overline{\alpha}_S.\overline{\iota}_S.LC \oplus \texttt{false}_V\iota.LOOP))\backslash\{\alpha_S,\iota_S,\alpha_C,\iota_C\}$$

where **S** and **C** are variables that are assumed to be well-behaved machines. The generic creates a local variable to hold the result of the evaluation of the conditional expression, and then recursively executes the conditional expression and sequence of statements until the conditional tests false.

For the actual loop,The recursive semantics for the loop is then given by:

E(⟦ **while** <condition>
loop
<sequence of statements>
end loop; ⟧, **LD**, **DE**, *T***)**

= **New(*LOOP*(** **E(**⟦<condition>⟧, **LD**, **DE**, *T***)**,
E(⟦<sequence of statements>⟧, **LD**, **DE**, *T***)))**

If we take advantage of the ability of machines to be repeatedly activated, we can use the following non-recursive (for the conditional and statement machines) formulation.

E(⟦ **while** <condition>
loop
<sequence of statements>
end loop; ⟧, **LD**, **DE**, *T***)**

= (LOOP | **V** | $\mathbf{M}_{COND}$ | $\mathbf{M}_{SOS}$) \{α_{SOS},ι_{SOS},α_{COND},ι_{COND}} ∪ *Sort*(**V**)

where

$$\text{LOOP} \equiv \alpha.\mathsf{LC}[\overline{\alpha}_{SOS}/\overline{\alpha}_S, \overline{\iota}_{SOS}/\overline{\iota}_S, \overline{\alpha}_{COND}/\overline{\alpha}_C, \overline{\iota}_{COND}/\overline{\iota}_C]$$

$$\mathbf{M}_{COND} \equiv \langle =^{\mathbf{COND}\to\mathbf{V}} \mid \mathbf{E}(\llbracket \texttt{<condition>} \rrbracket, \mathbf{LD}, \mathbf{DE}, T)\rangle$$

$$\mathbf{M}_{SOS} \equiv \mathbf{E}(\llbracket \texttt{<sequence of statements>} \rrbracket, \mathbf{LD}, \mathbf{DE}, T)$$

where SOS is the sequence of statements, COND is the machine that does the assignment of the conditional value to the variable **V**, and

$$\mathsf{LC} \equiv \overline{\alpha}_C.\overline{\iota}_C.(\texttt{true}_V\overline{\alpha}_S.\overline{\iota}_S.\mathsf{LC} \oplus \texttt{false}_V\iota.\text{LOOP})$$

5.9 Declarative Blocks

A declarative block is a collection of declarations followed by a sequence of statements and possibly an exception handler. There are a number of possible semantics for declarative blocks, each reflecting a different visibility semantics for the declarations that occur in the block. We show two possibilities here.

For let or let* visibility semantics, we have:

E(⟦ *blockname*:

```
declare
   <declarative part>
begin
   <sequence of statements>
exception
   <exception handler list>
end
```

⟧, **LD**, **DE**, *T***)**

$$= \mathbf{M} \equiv \langle\ \mathbf{E(}[\![\texttt{<declarative part>}]\!]\mathbf{,\ M{:}LD,\ DE,}\ T\mathbf{)}\ |$$
$$\mathbf{E(}[\![\texttt{<sequence of statements>}]\!]\mathbf{,\ M{:}LD,\ M{:}DE,}\ T\mathbf{)}\ |$$
$$\mathbf{E(}[\![\texttt{<exception handler list>}]\!]\mathbf{,\ M{:}LD,\ M{:}DE,}\ T\mathbf{)}\ \rangle$$

Side effects:

$$\mathbf{LD} = \mathbf{LD} \cup \{(\mathit{blockname}, \mathbf{M})\}$$

$$\mathbf{M{:}DE} = \mathbf{M{:}LD} \underset{\mathrm{m}}{\cup} \mathbf{DE}$$

For letrec visibility semantics the direct environment passed to the declarative part would be different, giving:

$$\mathbf{M} \equiv \langle\ \mathbf{E(}[\![\texttt{<declarative part>}]\!]\mathbf{,\ M{:}LD,\ M{:}DE,}\ T\mathbf{)}\ |$$
$$\mathbf{E(}[\![\texttt{<sequence of statements>}]\!]\mathbf{,\ M{:}LD,\ M{:}DE,}\ T\mathbf{)}\ |$$
$$\mathbf{E(}[\![\texttt{<exception handler list>}]\!]\mathbf{,\ M{:}LD,\ M{:}DE,}\ T\mathbf{)}\ \rangle$$

5.9.1 Declarative Part

The elaboration of the declarative part simply elaborates each of the declarations, composing any machines that result in parallel. It is important to note that the elaboration of a variable will return a machine. The elaboration of a function declaration or type declaration *will not return a machine* - the created machine will be associated with the declared name in the local declaration set **LD**, but no machine is actually instantiated as part of the elaboration.

For let or letrec visibility semantics, we would have

$$\mathbf{E(}[\![\texttt{<declarative part>}]\!]\mathbf{,\ LD,\ DE,}\ T\mathbf{)}$$

$$= \langle\ \mathbf{E}(\llbracket \texttt{<declaration}_1\texttt{>} \rrbracket, \mathbf{LD}, \mathbf{DE}, T) \mid$$
$$\mathbf{E}(\llbracket \texttt{<declaration}_2\texttt{>} \rrbracket, \mathbf{LD}, \mathbf{DE}, T) \mid \ldots \mid$$
$$\mathbf{E}(\llbracket \texttt{<declaration}_n\texttt{>} \rrbracket, \mathbf{LD}, \mathbf{DE}, T) \rangle$$

It is important to note that for letrec visibility semantics, the enclosing declarative block has included **LD** in the computation of **DE**. Thus the elaboration of one declaration could well affect the meaning of a reference in another. This points out the importance of the order of elaboration in determining the meaning of a program. Some languages put such severe constraints upon the relative positions of declarations with respect to references that the order of elaboration is not an issue. Other languages, like Ada, provide mechanisms to specify the order of elaboration in cases where the order may not be sufficiently constrained[1]. In [Brow90] we show that an appropriate ordering, if one exists, may be determined through the construction of a dependency graph relating declarations and references. Cycles in this graph indicate that no proper elaboration ordering exists.

For let* visibility semantics, we would have

$$\mathbf{E}(\llbracket \texttt{<declarative part>} \rrbracket, \mathbf{LD}, \mathbf{DE}_0, T)$$

$$= \langle\ \mathbf{E}(\llbracket \texttt{<declaration}_1\texttt{>} \rrbracket, \mathbf{LD}_1, \mathbf{DE}_0, T) \mid$$
$$\mathbf{E}(\llbracket \texttt{<declaration}_2\texttt{>} \rrbracket, \mathbf{LD}_2, \mathbf{DE}_1, T) \mid \ldots \mid$$
$$\mathbf{E}(\llbracket \texttt{<declaration}_n\texttt{>} \rrbracket, \mathbf{LD}_n, \mathbf{DE}_{n-1}, T) \rangle$$

Here the order of elaboration is defined to be the order of declaration. For these elaborations, we have:

$$\mathbf{LD}_0 = \varnothing$$

1. [LRM] p. 10-11

Prior to the ith elaboration, we have:

$$\mathbf{LD}_i = \mathbf{LD}_{i-1}$$

$$\mathbf{DE}_i = \mathbf{LD}_{i-1} \cup \mathbf{DE}_0$$

After the ith elaboration, $\mathbf{LD}_i$ also contains the declaration resulting from the elaboration. After the last elaboration, we compute the returned set of local declarations:

$$\mathbf{LD} = \mathbf{LD}_n$$

5.9.2 Sequence of Statements

Because statements may have labels on them and references to them, elaboration order is important here as well. We show the semantics for letrec style visibility[1] (for sequential or let* visibility, the local and direct environments are computed exactly as for the let* declarative items).

$$\mathbf{E}(\llbracket \texttt{<sequence of statements>} \rrbracket, \mathbf{LD}, \mathbf{DE}, T)$$

$$= \langle\ \mathbf{E}(\llbracket \texttt{<statement}_1\texttt{>} \rrbracket, \mathbf{LD}, \mathbf{DE}, T) \mid$$

$$\mathbf{E}(\llbracket \texttt{<statement}_2\texttt{>} \rrbracket, \mathbf{LD}, \mathbf{DE}, T) \mid \ldots \mid$$

$$\mathbf{E}(\llbracket \texttt{<statement}_n\texttt{>} \rrbracket, \mathbf{LD}, \mathbf{DE}, T)\ \rangle$$

5.9.3 Mixing Declarations and Statements

It should be apparent from the semantics given in the previous sections for declarations and statements that there is no difference at all in their handling at this level of the semantics!

1. For sequential elaboration order and let* visibility, the computation of the local and direct environments is exactly the same as for the let* visibility of declarations.

Consequently, a semantics could easily be given for languages (like C and C++) that allow declarations and statements to be mixed.

5.10 Complex Data Types

5.10.1 Simple Record

The semantics of a simple record is nearly identical to the treatment of the declarative part of a declarative block. The major difference is that machines resulting from the elaboration of the declarative part here *will not be returned as the result of the elaboration*. Instead, they become the prototype machine for the new state type. An entry is made in the *StateType* relation mapping the new machine to itself (this identifying this machine as the prototype machine for the state type).

E(⟦ `type` *recordname* `is record`
`<declarative part>`
`end record;` ⟧**,***T***)**
= 0

Side effects:

M ≡ **E(**⟦`<declarative part>`⟧, **M**:**LD**, **DE**, *T***)**

LD = **LD** ∪ {(*recordname*, **M**)}

StateType = *StateType* ∪ (**M**,**M**)

We note that the components of the record can be referenced with the selected component notation described in section 4.8 on page 87.

5.10.2 Arrays

An array is a composition of machines (the elements of the array) and an index operation that returns a pointer to the indicated element of the array. We concentrate first on the index machine. Let ***T*** be the abstract type of the elements of the array, ***TIndexType*** be the state type of the index of machines of type ***T***, and ***ArrayIndexType*** be the state type of the index into the array.

We let $o1_I$ be the action that the index would take to indicate that it is pointing to the first machine in the array, $o2_I$ indicate that the second machine is being pointed to, and so on. We note that *AbstractIndex*(**M**) maps machine **M** into the action of ***TIndexType*** that indicates that the ***TIndexType*** machine is pointing to machine **M**. We let $\mathbf{M}_1$, $\mathbf{M}_2$, $\mathbf{M}_3$... be the elements of the array. We then define the interaction machine prototype $\mathbf{I}_A$ for an array with the following equations, letting I be the initial state:

$$\mathrm{I} \equiv \alpha.\mathrm{I}_0$$

$$\begin{aligned}\mathrm{I}_0 \equiv\ & o1_I AbstractIndex(\mathbf{M}_1).\mathrm{I}_0 \oplus \\ & o2_I AbstractIndex(\mathbf{M}_2).\mathrm{I}_0 \oplus 1 \\ & \ldots \\ & on_I AbstractIndex(\mathbf{M}_n).\mathrm{I}_0 \oplus 1 \\ & \iota.\mathrm{I}\end{aligned}$$

where n is the number of elements in the array. Thus the semantics of a simplified array declaration of fixed size would be:

E(⟦ **type** *arrayname* **is array** <indexExpression> **of** *typename*;⟧,
LD, **DE**, *T***)**

$= \mathbf{0}$

with the side effects doing all the work. We compute the size of the array, elaborating the index expression and allowing it to execute, placing the result in $\mathbf{V}_{temp}$:

$\mathbf{V}_{temp} \equiv \mathbf{E(}[\![\texttt{V}_{\texttt{temp}}\texttt{: integer;}]\!]\mathbf{, LD, DE,}\ T\mathbf{)}\ |$

$\langle \mathbf{E(}[\![\texttt{V}_{\texttt{temp}}\texttt{:= <indexExpression>;}]\!]\mathbf{, LD, DE,}\ T\mathbf{)}\ |\ \overline{\alpha}_{\mathbf{V}}.\overline{\iota}_{\mathbf{V}}.\mathbf{0}\ \rangle$

We next create the prototype machine for the actual array:

$\mathbf{M} \equiv \langle\ \mathbf{M}_1 \equiv \mathbf{New(}\boldsymbol{T}\mathbf{)}\ |$

$\mathbf{M}_2\ \equiv \mathbf{New(}\boldsymbol{T}\mathbf{)}\ |$

$|\ldots$

$|\mathbf{M}_n \equiv \mathbf{New(}\boldsymbol{T}\mathbf{)}\ |$

$\mathbf{I}_A\rangle\ \rangle \backslash Sort(\mathbf{V}_{temp})$

where $\mathbf{I}_A$ is as defined above, $\boldsymbol{T} = \mathbf{Ref(}\textit{typename}\mathbf{,DE)}$, and n is the value encoded in $\mathbf{V}_{temp}$ after the index expression is evaluated. We also have the following side effects:

$\mathbf{LD} = \mathbf{LD} \cup \{(\textit{arrayname}\mathbf{,}\ \mathbf{M})\}$

*StateT*ype = *StateT*ype $\cup$ (**M**,**M**)

*AbtractT*ype = *AbtractT*ype $\cup$ (**M**,***TAbstractIndex***)

*AbtractT*ype = *AbtractT*ype $\cup$ (**M**,***ArrayIndexType***)

5.10.3 Accessing Arrays

To access an array, we simply create a modified pointer machine of type ***TAbstract-Index***. This modified pointer initializes the variable $\mathbf{V}_P$ before activating the interaction

machine, and hides the variable $\mathbf{V}_P$ so that the resulting pointer acts like a constant pointer. The variable $\mathbf{V}_P$ is initialized by evaluating the index expression and then using the $\mathbf{I}_A$ part of the array to map the index expression result into the initial value for $\mathbf{V}_P$. The formal semantics is given by[1]:

$$\mathbf{E}(\llbracket \ \textit{arrayname}\ \texttt{[<indexExpression>]} \rrbracket, \mathbf{LD}, \mathbf{DE}, T)$$
$$= \langle\ \langle\ \mathbf{V}_P \mid =^{\mathbf{I}\to\mathbf{V}};\ \ \mathbf{I}_P\ \rangle \setminus \mathit{Sort}(\mathbf{V}_P) \mid$$
$$\mathbf{E}(\llbracket \texttt{<indexExpression>} \rrbracket, \mathbf{LD}, \mathbf{DE}, T)\ \rangle$$

where $\mathbf{V}_P$ and $\mathbf{I}_P$ are defined as they were for pointers, V in the expression $=^{\mathbf{I}\to\mathbf{V}}$ stands for $\mathbf{V}_P$, and **I = Ref(***arrayname***,DE)**

5.11 Programs and Subprograms

In this section, we shall describe a model of a subprogram as a state machine. We shall initially develop a call by value and return semantics for a machine that is as independent as possible from the caller of the subprogram. We shall see that the executable body of this machine is *strictly orthogonal* to the calling program. Later, we shall see that the introduction of nested scopes and/or the introduction of other parameter passing semantics essentially relax the orthogonality between the executable body of the subprogram and the caller, and we shall explore the implications of this loss of orthogonality.

We model our subprogram as a machine that can be conceptually divided into four parts: a part that provides local state storage, a part that copies the initial values into the local storage, a part that performs the actual computation associated with the subprogram, and a part

1. As with the first subprogram semantics given, this semantic assumes that if <indexExpression> is a function, then it is a call by value and result function. Alternate semantics, similar to those shown for subprograms, may be used to generalize to any type of subprogram call semantics.

that copies result values back to the caller. We can thus view the subprogram as a composition of machines:

$$\langle \mathbf{M}_S | \mathbf{M}_I | \mathbf{M}_E | \mathbf{M}_O \rangle \quad \text{Eq. 23}$$

where $\mathbf{M}_S$ is the local state, $\mathbf{M}_I$ is the machine that provides initialization of the local state, $\mathbf{M}_E$ is the machine that actually implements the computation, and $\mathbf{M}_O$ is the machine that copies the values back out. We note that $\mathbf{M}_I$ and $\mathbf{M}_O$ are usually pure interaction machines.

5.11.1 Call By Value and Result

In call by value and result, values of input parameters are copied from the calling program's actual parameters to the subprogram's formal parameters prior to executing the body of the subprogram. Upon completion of the execution, values are copied from the formal parameters back to the actual parameters. Furthermore, the internal actions of the machine are hidden, giving an overall composition that has the structure:

$$\langle \mathbf{M}_S | \mathbf{M}_I ; \mathbf{M}_E ; \mathbf{M}_O \rangle Sort(\mathbf{M}_S) \cup Sort(\mathbf{M}_E) \quad \text{Eq. 24}$$

In ore detail, we include a storage state machine for each formal parameter (including the implicit return formal parameter in a function) in the state machine for the subprogram. For each parameter of mode in, we add an assignment machine to copy the initial value into the formal parameter, and for each parameter of mode out, we add an assignment machine to copy the values back.

Consider the following Ada subprogram:

```
function F1 (a:integer) return integer is
   b: integer;
begin
```

```
      b := 2*a;
      return b;
   end F1;
```

This subprogram has three local declarations: the formal parameter `a`, the internal declaration `b`, and the unnamed return value that we shall call `r`. If we let S1 be the elaboration of the first statement, and S2 be the elaboration of the second, this subprogram would result in the following machine after elaboration:

$$\mathbf{F1} \equiv \langle \mathbf{M}_S | \mathbf{M}_I ; \mathbf{M}_E ; \mathbf{M}_O \rangle \restriction Sort(\mathbf{M}_I) \cup Sort(\mathbf{M}_O) \qquad \text{Eq. 25}$$

where

$$\mathbf{M}_S \equiv \langle \mathbf{V}_a \mid \mathbf{V}_b \mid \mathbf{V}_r \rangle$$

$$\mathbf{M}_I \equiv =^{in \to Va}$$

$$\mathbf{M}_E \equiv \langle \mathbf{C}_2 \mid \bigstar^{C2, \varsigma\alpha \to Vb} \rangle ; =^{Vb \to Vr} \rangle$$

$$\mathbf{M}_O \equiv =^{Vr \to out}$$

Expanding, we end up with:

$$\mathbf{F1} \equiv \langle \mathbf{V}_a \mid \mathbf{V}_b \mid \mathbf{V}_r \mid =^{in \to Va} ; \langle \mathbf{C}_2 \mid \bigstar^{C2, \varsigma\alpha \to Vb} \rangle ; =^{Vb \to Vr} ; =^{Vr \to out} \rangle \restriction Sort(\mathbf{in}) \cup Sort(\mathbf{out}) \qquad \text{Eq. 26}$$

where **in** and **out** are action sets associated with the input and output formal parameters.

A more traditional interpretation of the subprogram would be to not have local storage for the return value, but rather to interpret the return statement as a direct value copy to the target machine (with appropriate rearrangement of the synchronization constructs).

$$\mathbf{F2} \equiv \langle \mathbf{V}_a \mid \mathbf{V}_b \mid =^{in \to Va}; \langle \mathbf{C}_2 \mid \bigstar^{C2, \varsigma\alpha \to Vb} \rangle; =^{Vb \to out} \rangle \restriction Sort(\mathbf{in}) \cup Sort(\mathbf{out}) \qquad \text{Eq. 27}$$

As one might expect, this interpretation of the subprogram is behaviorally equivalent to the first.

5.11.2 Call by Reference

In call by reference, we dispense with the local storage for the formal parameters, and any assignment machines that were used to copy values in or out. In place of these mechanisms, we map the actions associated with the formal parameters to the actions associated with the actual parameters. Re-casting our earlier example in the call-by-reference form, we get:

$$\mathbf{F3} \equiv \langle \mathbf{V}_b \mid \langle \mathbf{C}_2 \mid \bigstar^{C2, \iota v \to Vb} \rangle; =^{Vb \to out} \rangle \restriction Sort(\mathbf{in}) \cup Sort(\mathbf{out}) \qquad \text{Eq. 28}$$

It should be noted that while this particular example is behaviorally equivalent to the call by value and return example, call by reference is not, in general, equivalent. Consider the following slightly altered program:

```
function G (a:integer) return integer is
   b: integer;
begin
   b := a*a;
   return b;
end F1;
```

$$\mathbf{G1} \equiv \langle \mathbf{V}_b \mid \bigstar^{in, \iota v \to Vb} \rangle; =^{Vb \to out} \rangle \restriction Sort(\mathbf{in}) \cup Sort(\mathbf{out}) \qquad \text{Eq. 29}$$

This machine will generate *two* $\mathtt{in}_i$ actions instead of the on that would occur with call by value and result and is, therefore, not observably equivalent to a call by value and result version of the same program.

6.0 Discussion and Conclusions

We have shown how the semantics of a programming language can be constructively given in terms of primitive state machines and compositions of state machines. Thus the semantics of a program is given as an abstract state machine whose structure is constructively specified by the program itself. We have shown that the dominant concepts of a programming language are readily understood in terms of three basic semantic concepts: state machines, state types (sets of isomorphic state machines), and generic machines (parameterized specifications of state types.) We have shown that programs, subprograms and data types all have a uniform interpretation as state types. We have described the relationship between the identifiers in the language and the semantic model elements that they correspond to, and we have provided a set-theoretic description of the computation of visibility in programming languages.

Constructive semantics is fully abstract in the sense that behavioral equivalence defines equivalence classes of semantic expressions, and these equivalence classes can be taken to be the fully abstract semantics of the expression. We have left two interesting questions open in this area. Is there a normal form for machine algebra expressions that would ease their syntactic comparison? Is behavioral equivalence decidable in the restricted classes of machines used in our semantics? Brookes' work on normal forms of synchronization trees[1] (which underlie HCCS) leads us to believe that for *finite* machines a unique (up to the ordering of terms) normal form exists in HCCS for each equivalence class, but we suspect that the existence of a normal form of machine algebra expressions is precluded by the constraints that our machine algebra places on the form of HCCS expressions.

1. [Broo83] pp. 99-100.

6.1 Topics for Future Work

There are a number of areas of the semantic modeling process that we have made a start in, but which would benefit from additional work. These areas include: a more complete exploration of the notion of aborting the execution of a machine; a fuller exploration of the type system's capability of modeling concepts of type, including variant record structures and type hierarchies arising in object oriented programming languages; an exploration of the use of generic state machines to model Ada packages, tasks and generics; a detailed exploration of the interplay between structured types and overload resolution; and consideration of extending our notion of behavioral equivalence to include Hoare's divergences equivalence. In the following we briefly discuss each of these areas.

6.1.1 Abort Actions

In our exploration of goto and exception semantics, we found a need for an abort action that would return a machine to its initial state without executing its idle action. We did not, however, explore the behavior of sequential and parallel compositions of machines when one of those machines aborts. In particular, we have left for future work a re-definition of a well-behaved machine to include the notion of abortion, a corresponding re-definition of our composition operators and the proof that the revised composition operators preserve the well-behaved property.

6.1.2 Modeling More Complex Type Structures

We have left for later work a description of the semantics of variant record types and the closely related description of class hierarchies in object oriented languages. It is our conjecture that a complete semantics for both can be given with the mechanisms described in this thesis. In particular, it is our belief that our definition of abstract types provide a suffi-

cient model for the “common” behavior of a parent class that is shared between its subclasses. Variant records are similar, with the shared portion being the discriminant (the portion of the record indicating which variant - subclass- is actually present in the current record.

6.1.3 Ada Packages, Tasks and Generics

We have left the semantics of Ada packages, tasks and generics for future work. As with more complex type structures, it is our conjecture that these three language constructs can be modeled with the mechanisms described in this thesis. In [Brow90] we have already explored the visibility computations in these language constructs. For Ada packages, it is the specification/body visibility rules alone that distinguish packages from the declarative blocks described in this thesis. For tasks, the notions of concurrency and synchronous rendezvous have been added, but we note that both of these concepts are already fundamental parts of our model. Consequently, we do not anticipate any problems in modeling tasks. Finally, we believe that generic types, as defined here, will provide an adequate model of Ada generics. Generics in Ada are parameterized packages, types and subprograms whose parameters are themselves types, subprograms or objects (variables). All of these parameter constructs are modeled in our semantics by machines, and our generic types have in them variables whose values are machines. Thus we anticipate that Ada generics will be modeled in a very straightforward manner with our generic types.

6.1.4 Overload Resolution

The overload resolution model presented in this thesis shares type information between references in a very limited manner. In [Brow89] we have explored more complicated overload resolution schemes sufficient to describe overload resolution as used in Ada, including the resolution of generic parameters. This work needs to be merged with the current model to provide a complete semantics for Ada overload resolution. Additional work of more fun-

damental interest is an extension of this work to consider the interplay between type networks (multiple inheritance in object oriented languages) and overload resolution. We have already demonstrated (in the pointer semantics) that multiple inheritance can be modeled in our type system.

6.1.5 Divergence

Hoare's failures equivalence, which we have used as our notion of behavioral equivalence, encompasses two concepts about the observable behavior of machines: the sequences of actions that it is capable of executing (its traces), and the actions that it will possibly refuse to take after performing a sequence of actions (its failures). Hoare [Hoar85] and Brookes [Broo83] observe that there is a third significant possibility, and that a machine may go into an infinite computation after performing a sequence of actions (its divergences). This we might construe as a situation in which the machine may refuse to take any observable action, but Hoare[1] has gives an interpretation in which such a machine is construed to be able to also take any possible action once it has diverged. We have been unable, as of yet, to arrive at a suitable interpretation of divergence in our model, and have left it out of our definition of behavioral equivalence. At some point the issue of the proper interpretation of divergence in constructive semantics should be resolved and incorporated into the notion of behavioral equivalence.

6.2 Benefits of Constructive Semantics

We believe that constructive semantics can be used in a very practical way by both compiler writers and programmers. If a compiler writer uses constructive semantics to specify the characteristics of the machines that the compiler *generates* (how it implements various lan-

1. [Hoar85] p. 130.

guage constructs), behavioral equivalence can be used to establish whether or not those generated machines are equivalent to the constructs given by the language specification (also given with constructive semantics). Furthermore, since behavioral equivalence is a congruence relation, this equivalence can be established by comparing the implementation and the language specification on a construct-by-construct basis, thus simplifying the complexity of the task.

From the programmer's perspective, we hope that programmers will find constructive semantic descriptions to be readily understandable. We believe that the abstraction mechanisms used in constructive semantics are very similar to those that the programmer uses every day in designing modules. This similarity should allow the programmer to reason about the constructive semantics in a manner similar to the way that he or she reasons about programs, first understanding the behavior of a module, and then later using that module in some other context. This, of course, remains to be demonstrated.

A HCCS

HCCS is the semantic model used in this thesis, and is a hybrid of work done by Milner, Hoare and Brookes. The formal basis for this model is the synchronization tree semantics of Brookes [Brook83].

The starting point for the semantic model is Milner's CCS, with the following modifications:

1 Instead of using CCS's flat set of actions, we use the abelian group of actions as used in Milner's SCCS and ASCCS.

2 Milner's + operator is replaced by the similar □ operation used by Hoare and defined in terms of Milner's synchronization trees by Brookes [Broo83]. We will continue to use + to represent this operation. The use of the Hoare operator makes failures equivalence a congruence relation.

3 We alter the definition of Milner's | operation to allow n-way synchronization between agents, where CCS only allows binary synchronization. We provide a formal semantics for this new operation using Brookes' synchronization tree semantics.

4 We use Brookes' failures equivalence rather than Milner's observational equivalence. Milner's observational equivalence makes non-observable distinctions between machines, while Brookes' failures equivalence only distinguishes between machines if the machines differ in observable behavior. The use of failures equivalence allows the proof of our parallelization theorem, which is not true using observational equivalence.

A.1 Actions

We first define Act, the set of ***actions***. Actions, as we shall see, can be thought of as the labels on the arcs of state machines. Let $A = \{..., \tilde{c}, \tilde{b}, \tilde{a}, 1, a, b, c, ...\}$ be a set of *primitive actions*, and let $\times$ be a binary operator and $\bar{\ }$ be a unary inverse operator. Then we define Act, the set of ***actions***, to be the set generated by the following rules:

$A \subset Act$

$\forall a, b \in Act:$

1) $a \times b \in Act$

2) $\tilde{a} \in Act$

In writing actions, we shall frequently omit the parallel composition operator $\times$, writing st to represent $s \times t$.

We add the following ***action axioms***:

$$a \times b = b \times a \tag{30}$$

$$(a \times b) \times c = a \times (b \times c) \tag{31}$$

$$a \times 1 = a \tag{32}$$

$$a \times \tilde{a} = 1 \tag{33}$$

From these axioms, it follows that:

$$\tilde{1} = 1 \tag{34}$$

$$\bar{\bar{a}} = a \tag{35}$$

Then we have:

$$(Act, \times, 1, \bar{\ }) \text{ is an Abelian Group} \tag{36}$$

We define *Atom(*a*)* to be the set of primitive actions that a is comprised of. For example, if a and b are primitive actions, then *Atom(*ab*)* = {a, b}. By extension, if A is a set of actions, we define $Atom(A) = \cup_{a \in A} Atom(a)$. For $A \subseteq Act$, we define A^+ to be the submonoid of Act freely generated by A.

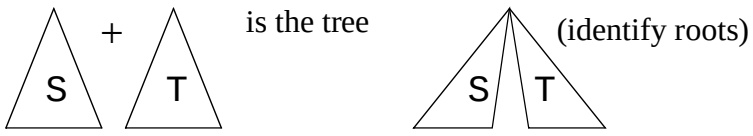
A.2 Synchronization Tree Semantics

The semantics of our model is given in terms of synchronization trees in the manner of Milner. We will here borrow the definitions and notation of Brookes [Broo83], noting deviations from Brookes as they occur.

We begin with a basic algebra over synchronization trees (ST's)[1] whose operations are NIL, +, and a(), where NIL is the trivial tree:

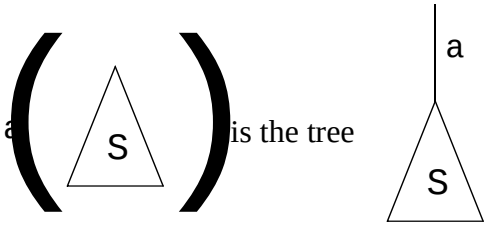
•

$+$[2] is a binary operation combining two trees:



1. [Broo83] pp. 91-92
2. This is Milner's + operation.

and $a(S)$ is a unary operation for each $a \in Act$:



Thus the general synchronization tree can be expressed in the form:

$$S = \sum_{i=1}^{n} a_i S_i + \sum_{i=1}^{m} 1 S'_j$$

where $a_i \in Act - \{1\}$.

In the case where S=NIL, we have n=m=0.

If *s* is a sequence of observable actions, we say that a subtree S' is an ***s-derivative*** of S if there exists a path from the root of S to the root of S' whose observable actions form the sequence *s*. Following Milner, we define the relation $S \stackrel{s}{\Rightarrow} S'$ between trees:

$S \stackrel{s}{\Rightarrow} S'$ iff S' is an *s*-derivative of S

We have the following laws for ST's, letting $S, T, U \in \boldsymbol{ST}$:

Axiom 6 Associativity of Synchronization Trees

$$S+(T+U) = (S+T)+U$$

Axiom 7 Commutativity of Synchronization Trees

$$S+T = T+S$$

Axiom 8 Nullity of Synchronization Trees

$$S+NIL = S$$

Axiom 9 Idempotence of Synchronization Trees

$$S+S = S$$

A.3 Agent Expressions and Agents

We now proceed to give a syntax for the specification of a state machine as a synchronization tree. Subtrees in the synchronization trees will correspond to syntactic constructs that we shall call agent expressions. We begin by defining $\boldsymbol{E}$, the set of ***agent expressions***. Each of these expressions can be thought of as representing a state of the machine. Agent expressions may contain variables, in which case the expression is a *template* for a state that will be fully specified when the values of the variables are specified. An ***agent*** is an agent expression that contains no variables. Such an expression, in conjunction with the inference rules that relate one agent expression (state) to the next, constitutes a fully specified state machine.

Formally, let $\boldsymbol{K} = \{\mathbf{0},\mathbf{1},A,B,C,...\}$ be a set of ***agent constants***[1], and $\boldsymbol{X} = \{X,Y,Z,...\}$ be a set of ***agent variables***. Let $\times$ and $+$ be binary composition operators. Let $\phi: Act \rightarrow Act$ be a mapping from Act to Act such that $\phi(a) = \overline{\phi(a)}$ and $\phi(1) = 1$. If A is a subgroup of

1. Constants are used to specify recursion. As we shall see shortly, constants are not strictly necessary, since the *fix* construct with variables is quite capable of representing recursive definitions. However, the use of constants to specify recursion makes the specifications somewhat easier to read.

Act, let $E \restriction A$ be a unary operator that restricts the actions of E to the actions present in A. Then we define $\boldsymbol{E}$, the set of ***agent expressions***, to be the set generated by the following rules:

$$\boldsymbol{K} \cup \boldsymbol{X} \subset \boldsymbol{E}$$

$$\forall\, E,F \in \boldsymbol{E},\, a \in Act:$$

Action:

1) $a.E \in \boldsymbol{E}$

Informally (we will give the formal semantics later), $a.E$ means that the agent expression $a.E$ performs the action a, and then performs the actions specified by E.

Product:

2) $E \mid F \in \boldsymbol{E}$

$E \mid F$ means that the agent expression is a composition of the agent expressions represented by E and F. Actions performed by the composition consist of either just one of the component elements (E or F) performing an action (*asynchronous* action) or both component elements (E and F) performing actions simultaneously (*synchronous* action).

Summation:

3) $\sum_{i \in I} E_i \in \boldsymbol{E}$

where E_i is an independent set of agent expressions from $\boldsymbol{E}$, and I is an index set. $\sum_{i \in I} E_i$ means that the agent expression represented by the summation is a composition of alternative agent expressions, and will ultimately perform the observable actions specified by exactly one of the expressions in the set of E_i. There is one summation that we shall have occasion to use frequently, and this is the ***inaction machine* 0**:

$$\mathbf{0} \equiv \sum_{i \in \varnothing} E_i \tag{37}$$

Because the set of expressions is empty, this machine never performs any actions.

In the special case of a summation involving two machines, we shall write the summation as $E \oplus F$.

It is important to note that the operation being defined here is *not* Milner's summation operator, but a generalization to an arbitrary number of terms of Hoare's $\Box$ operator. As we shall see, the behavior of the Milner and Hoare operators is the same for observable actions, but different for the non-observable action 1.

Restriction:

4) $E \upharpoonright A \in \boldsymbol{E}$

$E \upharpoonright A$ means that the agent expression is specified by E except that any actions not in $Atom(A)^*$ are hidden. If an action is hidden, then no agent expression outside of E may trigger that action. A related operation is to hide just the actions that are in $Atom(A)^*$. We shall designate this with the notation $E \backslash A$, and define this to be:

4a) $E \backslash A \equiv E \upharpoonright (Act - Atom(A)^*)$

Morphism:

5) $E[\phi] \in \boldsymbol{E}$

$E[\phi]$ means the agent expression specified by E after the actions have been mapped as specified by ϕ.

Recursion:

6) $\mathrm{fix}_j \tilde{\boldsymbol{X}} \tilde{\boldsymbol{E}} \in \boldsymbol{E}$

where $\tilde{\boldsymbol{X}} = \{X_i \mid i \in I\}$ is an I-indexed set of distinct agent expression variables, $\tilde{\boldsymbol{E}} = \{E_i \mid i \in I\}$ is an I-indexed set of agent expressions in which the variables X_i may occur free, and $j \in I$. The expression

$$\mathrm{fix}_j \tilde{\boldsymbol{X}} \tilde{\boldsymbol{E}}$$

represents the solution for the j^{th} variable from the set of equations $\{X_i \sim E_i : i \in I\}$, where the symbol ~ is a congruence relation called *strong bisimulation* that will be defined later. The prefix $\mathrm{fix}_j \tilde{\boldsymbol{X}}$ has the effect of binding *all* of the variables in $\tilde{\boldsymbol{X}}$.

The system of equations that appears in the recursion expressions is not unlike the production rules of a BNF grammar, where the expression on the right may be substituted for the variable that appears on the left wherever the variable appears. In the simplest case of only one variable, we omit the subscripts, giving $\mathrm{fix}\boldsymbol{XE}$.

Constant Definitions:

As an alternative to the use of the fix operation, we can use systems of equations involving constants, where a constant is one of the variables in a recursive set of equations, and the expression on the right hand side is one of the expressions on the right hand side in the recursive set of equations. Each constant is defined to be an agent expression E:

$$A \equiv E$$

A.4 Semantics of Agent Expressions

We give the semantics for agent expressions using Brookes' notation for synchronization trees. Where indicated, the semantic definitions are taken from [Broo83]. The mapping $[\![\bullet]\!]: \boldsymbol{E} \rightarrow \boldsymbol{ST}$ is given by:

$$[\![a.E]\!] = a([\![E]\!])$$

$$[\![E \mid F]\!] = [\![E]\!] \mid [\![F]\!]$$

$$[\![E \oplus F]\!] = [\![E]\!] \,\square\, [\![F]\!]$$

$$[\![E \restriction A]\!] = [\![E]\!] \restriction A$$

$$[\![E[\phi]]\!] = \phi([\![E]\!])$$

$$[\![A]\!] = [\![E]\!]$$

where $E, F \in \boldsymbol{E}$ are agent expressions, $A \in \boldsymbol{E}$ is a constant, and the tree operations $|$, $\square$, $\restriction$ and ϕ are given by:

$$S = \sum_{i=1}^{n} a_i S_i + \sum_{j=1}^{m} 1 S'_j$$

$$T = \sum_{j=1}^{N} b_j T_j + \sum_{j=1}^{M} 1 T'_j$$

$$S|T = \sum_{i=1}^{n} a_i (S_i|T) + \sum_{j=1}^{N} b_j (S|T_j) + \sum_{i=1}^{m} 1 (S'_i|T) + \sum_{j=1}^{M} 1 (S|T'_j)$$

$$+ \sum_{i=1}^{n}\sum_{j=1}^{N} a_i b_j (S_i|T_j) + \sum_{i=1}^{m}\sum_{j=1}^{M} 1 (S'_i|T'_j)$$ [1]

It should be recalled here that in the fifth term above, when $a_i = \bar{b}_j$, then $a_i b_j = 1$, and the resulting term becomes one of the silent transitions of the composition.

$$S \square A = \sum_{i=1}^{n} a_i S_i + \sum_{j=1}^{N} b_j T_j + \sum_{i=1}^{m} 1 (S'_i|T) + \sum_{j=1}^{M} 1 (S|T'_j)$$ [2]

$$S \upharpoonright A = \sum_{a_i \notin A} a_i (S_i \upharpoonright A) + \sum_{i=1}^{m} 1 (S'_i \upharpoonright A)$$

$$\phi(S) = \sum_{i=1}^{n} \phi(a_i)\,(\phi(S_i)) + \sum_{=1}^{m} 1 (\phi(S'_i))$$

1. This is slightly different than Brookes' definition for Milner's | operator, which only allows simultaneous action when $a_i = \bar{b}_j$

2. This is Brookes' semantics for Hoare's $\square$ operator.

A.5 Labeled Transition Systems

We now proceed to define a *labeled transition system*, which can be thought of as the specification of a state machine whose number of states is not necessarily finite. Agent expressions are the states of these machines, and actions label the transitions between agent expressions, hence the term labeled transition system. Formally, a ***labeled transition system*** is a triple:

$$(\boldsymbol{E},\ Act, \{\xrightarrow{a} : a \in Act\})$$

where $\boldsymbol{E}$ is a set of agent expressions, Act is a set of actions, and each $\xrightarrow{a}$ is a relation between agent expressions.

In this system, we have the following inference rules, derived from the synchronization tree semantics

Action:

$$\frac{}{a.E \xrightarrow{a} E} \quad [1] \qquad (38)$$

This rule says that the agent expression $a.E$ may be converted into the agent expression E when the action a occurs. (We note that the occurrence of an action means that some other agent expression, with which this one has been composed in a product, has simultaneously made a transition with the action a.)

Summation:

1. This is Brooke's (PRE) axiom of [Broo83] p. 168

$$\frac{E_j \xrightarrow{a} E'_j}{\sum_{i\in I} E_i \xrightarrow{a} E'_j} \quad (j\in I, a\neq 1)^{1} \tag{39}$$

This rule says that if an agent expression E_j can be converted into an agent expression E'_j when the observable action a occurs, then a summation $\sum_{i\in I} E_i$ containing E_j can also be converted into the agent expression E'_j when the action a occurs.

$$\frac{E_j \xrightarrow{1} E'_j}{\sum_{i\in I} E_i \xrightarrow{1} \sum_{i\in I} E_i[E'_j/E_j]} \quad (j\in I)^{2} \tag{40}$$

This rule says that if an agent expression E_j can be converted into an agent expression E'_j when the unobservable action 1 occurs, then a summation $\sum_{i\in I} E_i$ containing E_j can also be converted into the summation $\sum_{i\in I} E_i[E'_j/E_j]$ when the action 1 occurs.

This pair of rules has the effect of preventing a non-observable transition of one element of a summation from eliminating the other possibilities in the summation. This is distinctly different from Milner's semantics[3], which uses the first rule only and places no restriction on the action. The use of Milner's semantics prevents failures equivalence from being used as a congruence relation, which is a requirement of our model[4].

---

1. This is an obvious generalization of the ($COND_2$) inference rule of [Broo83] p. 168
2. This inference rule is both less restrictive and more general than the ($COND_1$) inference rule of [Broo83] p. 168
3. [Miln83] p. 271
4. Milner's observational equivalence is also not a congruence relation in CCS. It is conjectured that the use of Hoare's □ operator (which we are using here as our + operator) in lieu of Milner's + would make observational equivalence a congruence relation in the modified system.

Product:

$$\frac{E \xrightarrow{a} E'}{E \mid F \xrightarrow{a} E' \mid F} \tag{41}$$

$$\frac{F \xrightarrow{b} F'}{E \mid F \xrightarrow{b} E \mid F'} \tag{42}$$

These rules say that if either machine can make a transition, then that machine may make the transition in the product.

$$\frac{E \xrightarrow{a} E' \qquad F \xrightarrow{b} F'}{E \mid F \xrightarrow{ab} E' \mid F'} \tag{43}$$

This rule says that if the agent expression E can be converted into the agent expression E' when the action a occurs, and the agent expression F can be converted into the agent expression F' when the action b occurs, then the agent expression $E \mid F$ that is the product of these agent expressions can be converted into the agent expression $E' \mid F'$ when the action ab occurs (recall that ab is the product of the actions a and b).

Restriction:

$$\frac{E \xrightarrow{a} E'}{E \backslash S \xrightarrow{a} E' \backslash S} \quad (a \in S) \tag{44}$$

This rule says that if the agent expression E can be converted into the agent expression E' when the action a occurs, then the agent expression $E \upharpoonright S$ (which is the agent expression E with its actions restricted to the actions present in S) can be converted into the agent expression $E' \upharpoonright S$ when the action a occurs provided that a is a member of S. The implication is that if a is not a member of S then no transition is possible on a. Recall that the action 1 is always required to be a member of S.

Morphism:

$$\frac{E \xrightarrow{a} E'}{E[\phi] \xrightarrow{\phi(a)} E'[\phi]} \tag{45}$$

This rule says that if the agent expression E can be converted into the agent expression E' when the action a occurs, then the agent expression $E[\phi]$ (the agent expression E with all of its actions mapped into new actions by the morphism ϕ) can be converted into the agent expression $E'[\phi]$ by the action $\phi(a)$.

Recursion:

$$\frac{E_i[\mathrm{fix}\tilde{X}\tilde{E}/\tilde{X}] \xrightarrow{a} E'}{\mathrm{fix}_i\tilde{X}\tilde{E} \xrightarrow{a} E'} \tag{46}$$

This rule says that if the agent expression E_i can be converted into the agent expression E' when the action a occurs after all of the variables in $\tilde{X}$ have been bound to the agent expressions in $\tilde{E}$ and those bound values have been substituted for occurrences of the vari-

ables in E_i, then $\mathrm{fix}_i \tilde{X} \tilde{E}$ (which is the solution for the ith member of $\tilde{X}$) can also be converted into the agent expression E' when the action a occurs.

The equivalent rule for constants is:

$$\frac{E \xrightarrow{a} E'}{C \xrightarrow{a} E'} \quad (C \equiv E) \tag{47}$$

This rule says that if the constant C is defined to be the agent expression E, and E can make a transition to E' with the action a, then the constant C can also make a transition to E' with the action a.

A.6 Sorts of Agents

We now give some rules for determining the sort of an arbitrary agent.

We now define the *Sort()* operation, that takes as its argument an agent and returns the set of primitive actions that appear in the agent. The *Sort()* operation is defined recursively:

$$Sort(a.P) = Atom(a) \cup Sort(P) \tag{48}$$

$$Sort(\sum_{i \in I} P_i) = \cup_{i \in I} Sort(P) \tag{49}$$

$$Sort(P \mid Q) = Sort(P) \cup Sort(Q) \tag{50}$$

$$Sort(P \upharpoonright A) = Atom(A) \tag{51}$$

$$Sort(P \setminus A) = Sort(P) - A \tag{52}$$

$$Sort(\mathrm{fix}_i \tilde{\boldsymbol{X}} \tilde{\boldsymbol{E}}) = Sort(\tilde{\boldsymbol{X}}) \tag{53}$$

$$Sort(\mathsf{P}[\phi]) = Atom(\phi(Sort(\mathsf{P}))) \tag{54}$$

We also have the following theorems:

$$\mathsf{P} :: L \text{ and } L \subseteq M \text{ implies } \mathsf{P} :: M \tag{55}$$

$$\mathsf{a} \in L^{+} \text{ and } \mathsf{P} :: L \text{ implies } \mathsf{a} : \mathsf{P} :: L \tag{56}$$

$$\tilde{\boldsymbol{X}} :: L \text{ implies } \sum \tilde{\boldsymbol{X}} :: L \tag{57}$$

$$\mathsf{P} :: L \text{ and } \mathsf{Q} :: L \text{ implies } \mathsf{P} \mid \mathsf{Q} :: L \tag{58}$$

$$\mathsf{P} :: L \text{ implies } \mathsf{P} \backslash A :: (L - A) \tag{59}$$

$$\mathsf{P} :: L \text{ implies } \mathsf{P}[\phi] :: Atom(\phi(L)) \tag{60}$$

A.7 Failures Equivalence

Hoare characterizes a machine in terms of its *observable actions*, its *traces* (observable sequences of actions) and its *failures* (the actions that may not be responded to after a given sequence of observable actions).

The ***initials*** of a synchronization tree are the first observable actions of the tree. The ***traces*** of the tree are the sequences of observable actions that occur on any path beginning at the root of the tree. Equivalently, the traces of a tree S are the observable action sequences *s* for which S has an *s*-derivative. A tree can ***refuse*** a set of events X if it can make a silent transition to a subtree none of whose initials is a member of X[1]. The ***failures*** of a tree are

the pairs (s, X) such that the tree has an s-derivative that can refuse X. The following definitions[1] make this precise

$$\mathit{Initials}(S) = \{a:\in \mathit{Act} \mid \exists S' \text{s.t. } S \overset{\langle a \rangle}{\Rightarrow} S'\}$$

$$\mathit{Traces}(S) = \{s \mid \exists S' \text{s.t. } S \overset{s}{\Rightarrow} S'\}$$

$$\mathit{Refusals}(S) = \{X \mid \exists S' \text{s.t. } S \overset{\langle\rangle}{\Rightarrow} S' \text{ and } X \cap \mathit{Initials}(S') = \varnothing\}$$

$$\mathit{Failures}(S) = \{(s,X) \mid \exists S' \text{s.t. } S \overset{s}{\Rightarrow} S' \text{ and } X \cap \mathit{Initials}(S') = \varnothing\}$$

Brookes formally defines failure equivalence in terms of synchronization trees, giving us the following axioms and inference rule on synchronization trees, with $S, T, U \in \mathbf{ST}$[2]:

Axiom 10 Failure Equivalence of Synchronization Trees - B1

$$1S = S$$

Axiom 11 Failure Equivalence of Synchronization Trees - B2

$$S+1T+U = 1(S+T)+1T+U$$

Axiom 12 Failure Equivalence of Synchronization Trees - B3

$$aS+aT+U = a(1S+1T)+U$$

1. Note that every tree can make a silent transition to itself
1. [Broo83] p. 96
2. [Broo83] p. 98

Axiom 13 Failure Equivalence of Synchronization Trees - B4

$$1(aS+T)+1(aS'+T') = 1(aS+aS'+T)+1(aS+aS'+T')$$

Inference Rule 14 Failure Equivalence of Synchronization Trees - R

$$\frac{S = S'}{aS + T = aS' + T}$$

Now we must show that the tree operations the tree operations $a()$, $|$, $\oplus$, $\upharpoonright$ and ϕ respect failures equivalence. Brookes shows that $a()$, $\oplus$ and ϕ respect failures equivalence[1], so it remains to be shown that $|$ and $\upharpoonright$ respect failures equivalence.

Theorem 15 Composition Respects Failures Equivalence

For all $S, S', T, T' \in \boldsymbol{ST}$, $S=S'$ and $T=T'$ imply $S|T=S'|T'$

Before proceeding with the proof of this theorem[2], we must establish some supporting result. The traces of $S|T$ are obtained by composing the traces of S and T, in the following sense:

We define the set of all ***compositions*** $Comp(s,t)$ of two traces s and t by induction on the length of the traces:

1 $Comp(\langle\rangle,t) = Comp(t,\langle\rangle) = \{\langle t\rangle\}$

1. [Broo83] p106, theorem 4.3.4. Our $\oplus$ operation is Brookes' $\Box$ operation on trees, and our ϕ is Brookes' $f(S)$
2. Our proof is patterned after Brookes' proof for Milner's original | operation from CCS as found in [Broo83] p.106

2 $Comp(\langle a\rangle s,\langle b\rangle t) = \{\langle a\rangle u \mid u \in Comp(s,\langle b\rangle t)\} \cup \{\langle b\rangle u \mid u \in Comp(\langle a\rangle s,t)\}$

$\cup \{\langle ab\rangle u \mid a \neq b,\ u \in Comp(s,t)\} \cup \{u \mid a = b,\ u \in Comp(s,t)\}$

Lemma 16 Traces of a Composition

The traces of a composition $S|T$ are obtained by the compositions of their traces:

$$Traces(S|T) = \bigcup_{s,t} Comp(s, t) \quad \text{where } s \in Traces(S) \text{ and } t \in Traces(T)$$

Proof: by inspection of the definition of the | operation on trees.

Lemma 17 Initials of a Composition

The initials of a composition $S|T$ are obtained from the initials of S and T:

$$Initials(S|T) = Initials(S) \cup Initials(T)$$
$$\cup \{st \mid s \in Initials(S),\ t \in Initials(T),\ s \neq t\}$$
$$\bigcup_{s,t} Initials(S') \cup Initials(T')$$
$$\text{where } s,t \in \{(s,t) \mid \exists S' \text{ s.t. } S \stackrel{s}{\Rightarrow} S' \text{ and } \exists T' \text{ s.t. } T \stackrel{t}{\Rightarrow} T'\}$$

Proof: by inspection of the definition of the | operation on trees.

Lemma 18 Refusals of a Composition

The refusals of a composition $S|T$ are obtained from the refusals of S and T:

$$Refusals(S|T) = \{x \mid x \in Refusals(S) \text{ and } x \in Refusals(T)\}$$
$$\cup \{st \mid s \in Refusals(S) \text{ and } t \in Refusals(T)\}$$
$$\cup \{x \mid \exists s \text{ s.t. } S \stackrel{s}{\Rightarrow} S' \text{ and } T \stackrel{\bar{s}}{\Rightarrow} T' \text{ and } x \in Refusals(S') \text{ and } x \in Refusals(T')\}$$

Proof: by inspection of the definition of the | operation on trees.

Lemma 19 Failures of a Composition

$$Failures(S|T) = \{(s,X) \mid s \in \bigcup_{s,t} Comp(s,t) \text{ where } s \in Traces(S) \text{ and } t \in Traces(T), \text{ and}$$
$$X = \{x \mid x \in Refusals(S/s) \text{ and } x \in Refusals(T/t)\}$$
$$\cup \{st \mid s \in Refusals(S/s) \text{ and } t \in Refusals(T/t)\}$$
$$\cup \{x \mid \exists u \text{ s.t. } S/s \stackrel{u}{\Rightarrow} S' \text{ and } T/t \stackrel{\bar{u}}{\Rightarrow} T' \text{ and } x \in Refusals(S') \text{ and } x \in Refusals(T')\}$$

Proof: by inspection of the definition of failures and Lemma 16, Lemma 17 and Lemma 18.

Proof of Theorem 15: By the previous three lemmas, the failures of the composition are uniquely determined by the failures of the component trees. Since the theorem replaces each of the subtrees with failure equivalent trees, their failures must be identical, and the failures of the composition must be the same. Q.E.D.

Theorem 20 Restriction Respects Failures Equivalence

For all $S, S' \in \boldsymbol{ST}$, $S=S'$ implies $S \restriction A = S' \restriction A$

Proof: similar to Theorem 15

A.8 Equational Properties

Let $\boldsymbol{P}$ be the set of agents. Then for $P,Q,R \in \boldsymbol{P}$, the following theorems are valid:

$$P \mid Q = Q \mid P \tag{61}$$

$$(P \mid Q) \mid R = P \mid (Q \mid R) \tag{62}$$

$$P \restriction Sort(P) = P \tag{63}$$

$$Sort(P) \subseteq Atom(S) \text{ implies } (P \mid Q) \restriction S = P \mid Q \restriction S \tag{64}$$

$$S \perp Sort(P) \text{ implies } (P \mid Q) \setminus S = P \mid Q \setminus S \tag{65}$$

$$S \perp Sort(P) \text{ implies } P \setminus S = P \tag{66}$$

$$(P \restriction S) \restriction T = P \restriction (S \cap T) \tag{67}$$

$$(P \setminus S) \setminus T = P \setminus (S \cup T) \tag{68}$$

$$\mathbf{0} \mid \mathbf{0} = \mathbf{0} \tag{69}$$

B Machine Algebra

While our formal semantics (App. A) is based upon the concepts of states and transitions, we would like instead to talk in terms of *entire machines*. To emphasize this distinction, we use a bold symbol $\mathbf{M}$ to represent a machine. To transform a machine expression into a CCS-like state expression, we simply replace each machine $\mathbf{M}$ with the first state $\mathbf{M}_0$ of the defining expression for $\mathbf{M}$ *or with the first state of any observationally equivalent machine*. This conveniently allows us to conduct proofs using the calculus developed in App. A, and convert back to machine expressions at will.

Theorem 21 Commutivity

$$\forall \ \mathbf{M}_j, \mathbf{M}_k$$

$$\mathbf{M}_j \mid \mathbf{M}_k = \mathbf{M}_k \mid \mathbf{M}_j$$

By App. A (61).

Theorem 22 Associativity:

$$\forall \ \mathbf{M}_j, \mathbf{M}_k, \mathbf{M}_m$$

$$(\mathbf{M}_j \mid \mathbf{M}_k) \mid \ \mathbf{M}_m = \mathbf{M}_k \mid (\mathbf{M}_j \mid \mathbf{M}_m)$$

By App. A (62)

Theorem 23 Orthogonality and Symbol Hiding

$$\forall \ \mathbf{M}_j, \mathbf{M}_k$$

$$\mathbf{M}_j \ \perp \ \mathbf{M}_k \backslash Sort(\mathbf{M}_k)$$

1: $Sort(\mathbf{M}_k \backslash Sort(\mathbf{M}_k)) = \{1\}$ definition of \

2: $Sort(\mathbf{M}_j) \cap \{1\} = \{1\}$ definition of $\cap$

3: $Sort(\mathbf{M}_j) \cap Sort(\mathbf{M}_k \backslash Sort(\mathbf{M}_k)) = \{1\}$ 1 & 2

4: $\mathbf{M}_j \perp \mathbf{M}_k \backslash Sort(\mathbf{M}_k)$ 3 & definition of $\perp$

Q.E.D.

Theorem 24 Elimination of Actions

$$(b.\mathbf{M}_j \mid \bar{b}.\mathbf{M}_k)\backslash\{b\} = (\mathbf{M}_j \mid \mathbf{M}_k)\backslash\{b\}$$

We show that the two sides have the same semantics

1: $[\![(b.\mathbf{M}_j \mid \bar{b}.\mathbf{M}_k)\backslash\{b\}]\!] = ([\![b.\mathbf{M}_j \mid \bar{b}.\mathbf{M}_k]\!])\backslash\{b\}$ Semantics of \

2: $[\![b.\mathbf{M}_j \mid \bar{b}.\mathbf{M}_k]\!] = [\![b.\mathbf{M}_j]\!] \mid [\![\bar{b}.\mathbf{M}_k]\!]$ Semantics of |

3: $[\![b.\mathbf{M}_j]\!] = b([\![\mathbf{M}_j]\!])$ Semantics of .

4: $[\![\bar{b}.\mathbf{M}_k]\!] = \bar{b}([\![\mathbf{M}_k]\!])$ Semantics of .

5: $[\![b.\mathbf{M}_j \mid \bar{b}.\mathbf{M}_k]\!] = b([\![\mathbf{M}_j]\!]) \mid \bar{b}([\![\mathbf{M}_k]\!])$ 2, 3 & 4

6: $[\![b.\mathbf{M}_j \mid \bar{b}.\mathbf{M}_k]\!] = b([\![\mathbf{M}_j]\!]) + \bar{b}([\![\mathbf{M}_k]\!]) + 1([\![\mathbf{M}_j]\!] \mid [\![\mathbf{M}_k]\!])$

Definition of | for ST's

7: $[\![(b.\mathbf{M}_j \mid \bar{b}.\mathbf{M}_k)\backslash\{b\}]\!] = (b([\![\mathbf{M}_j]\!]) + \bar{b}([\![\mathbf{M}_k]\!]) + 1([\![\mathbf{M}_j]\!] \mid [\![\mathbf{M}_k]\!]))\backslash\{b\}$

1 & 6

8: $[\![(b.\mathbf{M}_j \mid \bar{b}.\mathbf{M}_k)\backslash\{b\}]\!] = (1([\![\mathbf{M}_j]\!] \mid [\![\mathbf{M}_k]\!]))\backslash\{b\}$ Definition of \ for ST's

9: $[\![(b.M_j \mid \bar{b}.M_k)\backslash\{b\}]\!] = ([\![M_j]\!] \mid [\![M_k]\!])\backslash\{b\}$ 8 & App. A Axiom 10

10: $[\![M_j]\!] \mid [\![M_k]\!] = [\![M_j \mid M_k]\!]$ Semantics of |

11: $[\![(b.M_j \mid \bar{b}.M_k)\backslash\{b\}]\!] = ([\![M_j \mid M_k]\!])\backslash\{b\}$ 9 & 10

12: $[\![(b.M_j \mid \bar{b}.M_k)\backslash\{b\}]\!] = [\![(M_j \mid M_k)\backslash\{b\}]\!]$ 11 & Semantics of \

Q.E.D.

Theorem 25 Elimination of Actions

$$(ab.M_j \mid \bar{b}c.M_k)\backslash\{b\} = ac.(M_j \mid M_k)\backslash\{b\}$$

We show that the two sides have the same semantics

1: $[\![(ab.M_j \mid \bar{b}c.M_k)\backslash\{b\}]\!] = ([\![ab.M_j \mid \bar{b}c.M_k]\!])\backslash\{b\}$ Semantics of \

2: $[\![ab.M_j \mid \bar{b}c.M_k]\!] = [\![ab.M_j]\!] \mid [\![\bar{b}c.M_k]\!]$ Semantics of |

3: $[\![ab.M_j]\!] = ab([\![M_j]\!])$ Semantics of .

4: $[\![\bar{b}c.M_k]\!] = \bar{b}c([\![M_k]\!])$ Semantics of .

5: $[\![ab.M_j \mid \bar{b}c.M_k]\!] = ab([\![M_j]\!]) \mid \bar{b}c([\![M_k]\!])$ 2, 3 & 4

6: $[\![ab.M_j \mid \bar{b}c.M_k]\!] = ab([\![M_j]\!]) + \bar{b}c([\![M_k]\!]) + ac([\![M_j]\!] \mid [\![M_k]\!])$

Definition of | for ST's

7: $[\![(ab.M_j \mid \bar{b}c.M_k)\backslash\{b\}]\!] = (ab([\![M_j]\!]) + \bar{b}c([\![M_k]\!])$

$+ ac([\![M_j]\!] \mid [\![M_k]\!]))\backslash\{b\}$ 1 & 6

8: $[\![(\mathtt{ab}.\mathbf{M}_j \mid \bar{\mathtt{b}}\mathtt{c}.\mathbf{M}_k)\backslash\{\mathtt{b}\}]\!] = (\mathtt{ac}([\![\mathbf{M}_j]\!] \mid [\![\mathbf{M}_k]\!]))\backslash\{\mathtt{b}\}$

Definition of \ for ST's

9: $[\![\mathbf{M}_j]\!] \mid [\![\mathbf{M}_k]\!] = [\![\mathbf{M}_j \mid \mathbf{M}_k]\!]$ Semantics of |

10: $[\![(\mathtt{b}.\mathbf{M}_j \mid \bar{\mathtt{b}}.\mathbf{M}_k)\backslash\{\mathtt{b}\}]\!] = (\mathtt{ac}([\![\mathbf{M}_j \mid \mathbf{M}_k]\!]))\backslash\{\mathtt{b}\}$ 8 & 9

11: $[\![(\mathtt{b}.\mathbf{M}_j \mid \bar{\mathtt{b}}.\mathbf{M}_k)\backslash\{\mathtt{b}\}]\!] = ([\![\mathtt{ac}.(\mathbf{M}_j \mid \mathbf{M}_k)]\!])\backslash\{\mathtt{b}\}$

10 & Semantics of .

12: $[\![(\mathtt{b}.\mathbf{M}_j \mid \bar{\mathtt{b}}.\mathbf{M}_k)\backslash\{\mathtt{b}\}]\!] = [\![(\mathtt{ac}.(\mathbf{M}_j \mid \mathbf{M}_k))\backslash\{\mathtt{b}\}]\!]$ 11 & Semantics of \

Q.E.D.

Theorem 26 Associativity of ;

$$(\mathbf{M}_i; \mathbf{M}_j);\mathbf{M}_k = \mathbf{M}_i;(\mathbf{M}_j;\mathbf{M}_k)$$

1: $(\mathbf{M}_i; \mathbf{M}_j);\mathbf{M}_k = ((\mathbf{M}_i \mid \mathbf{A}^{i\to j} \mid \mathbf{M}_j)\backslash\{\iota_i,\alpha_j\} \mid \mathbf{A}^{j\to k} \mid \mathbf{M}_k)\backslash\{\iota_j,\alpha_k\}$

Definition of ;

2: $\mathbf{A}^{j\to k} \perp \{\iota_i,\alpha_j\}$ Definition of $\mathbf{A}^{j\to k}$

3: $\mathbf{M}_k \perp \{\iota_i,\alpha_j\}$ Uniqueness of ι and α

4: $(\mathbf{M}_i; \mathbf{M}_j);\mathbf{M}_k = ((\mathbf{M}_i \mid \mathbf{A}^{i\to j} \mid \mathbf{M}_j \mid \mathbf{A}^{j\to k} \mid \mathbf{M}_k)\backslash\{\iota_i,\alpha_j\})\backslash\{\iota_j,\alpha_k\}$

1, 2, 3 & App. A (65)

5: $(\mathbf{M}_i; \mathbf{M}_j);\mathbf{M}_k = (\mathbf{M}_i \mid \mathbf{A}^{i\to j} \mid \mathbf{M}_j \mid \mathbf{A}^{j\to k} \mid \mathbf{M}_k)\backslash\{\iota_i,\alpha_j,\iota_j,\alpha_k\}$ App. A (68)

6: $\mathbf{M}_i \perp \{\iota_j,\alpha_k\}$ Uniqueness of ι and α

7: $\mathbf{A}^{i\to j} \perp \{\iota_j, \alpha_k\}$ Definition of $\mathbf{A}^{i\to j}$

8: $(\mathbf{M}_i; \mathbf{M}_j);\mathbf{M}_k = (\mathbf{M}_i \mid \mathbf{A}^{i\to j} \mid (\mathbf{M}_j \mid \mathbf{A}^{j\to k} \mid \mathbf{M}_k)\backslash\{\iota_j,\alpha_k\})\backslash\{\iota_i,\alpha_j\}$

5,6,7 & App. A (65) & App. A (68)

9: $(\mathbf{M}_i; \mathbf{M}_j);\mathbf{M}_k = \mathbf{M}_i;(\mathbf{M}_j;\mathbf{M}_k)$ Definition of ;

Q.E.D.

Theorem 27 Removal of brackets

$$\langle \mathbf{M}_j \mid \langle \mathbf{M}_k \rangle \rangle = \langle \mathbf{M}_j \mid \mathbf{M}_k \rangle$$

1: $\langle \mathbf{M}_j \mid \langle \mathbf{M}_k \rangle \rangle =$

$((\alpha_x \overline{\alpha}_j \overline{\alpha}_y .)^* \mid \mathbf{M}_j \mid ((\alpha_y \overline{\alpha}_k .)^* \mid \mathbf{M}_k \mid (\overline{\iota}_k \iota_y .)^*)\backslash\{\alpha_k,\iota_k\} \mid (\overline{\iota}_j \overline{\iota}_y \iota_x .)^*)\backslash\{\alpha_j,\alpha_y,\iota_j,\iota_y\}$

Definition of $\langle\rangle$

2: $(\alpha_x \overline{\alpha}_j \overline{\alpha}_y .)^* \perp \{\alpha_k,\iota_k\}$, $\mathbf{M}_j \perp \{\alpha_k,\iota_k\}$, $(\overline{\iota}_j \overline{\iota}_y \iota_x .)^* \perp \{\alpha_k,\iota_k\}$ Uniqueness of ι and α

3: $\langle \mathbf{M}_j \mid \langle \mathbf{M}_k \rangle \rangle =$

$((\alpha_x \overline{\alpha}_j \overline{\alpha}_y .)^* \mid \mathbf{M}_j \mid (\alpha_y \overline{\alpha}_k .)^* \mid \mathbf{M}_k \mid (\overline{\iota}_k \iota_y .)^* \mid (\overline{\iota}_j \overline{\iota}_y \iota_x .)^*)\backslash\{\alpha_k,\iota_k,\alpha_j,\alpha_y,\iota_j,\iota_y\}$

1,2 & App. A (65) & App. A (68)

4: $(\mathbf{M}_k \mid (\overline{\iota}_k \iota_y .)^* \mid (\overline{\iota}_j \overline{\iota}_y \iota_x .)^*) \perp \{\alpha_y\}$, $((\alpha_x \overline{\alpha}_j \overline{\alpha}_y .)^* \mid (\alpha_y \overline{\alpha}_k .)^* \mid \mathbf{M}_k) \perp \{\iota_y\}$

Uniqueness of ι and α

5: $\langle \mathbf{M}_j \mid \langle \mathbf{M}_k \rangle\rangle =$

$$(((\alpha_x\overline{\alpha_j}\overline{\alpha_y}.)^* \mid (\alpha_y\overline{\alpha_k}.)^*)\backslash\{\alpha_y\} \mid \mathbf{M}_j \mid \mathbf{M}_k \mid ((\overline{\iota_k}\iota_y.)^* \mid (\overline{\iota_j}\overline{\iota_y}\iota_x.)^*)\backslash\{\iota_y\})\backslash\{\alpha_k,\iota_k,\alpha_j,\iota_j\}$$

3,4 & App. A (65) & App. A (68)

6: $((\alpha_x\overline{\alpha_j}\overline{\alpha_y}.)^* \mid (\alpha_y\overline{\alpha_k}.)^*)\backslash\{\alpha_y\} = ((\alpha_x\overline{\alpha_j}\overline{\alpha_k}.((\alpha_x\overline{\alpha_j}\overline{\alpha_y}.)^* \mid (\alpha_y\overline{\alpha_k}.)^*))\backslash\{\alpha_y\}$

Theorem 25, with $a = \overline{\alpha_k}$, $b = \alpha_y$, $c = \alpha_x\overline{\alpha_j}$

7: $((\alpha_x\overline{\alpha_j}\overline{\alpha_y}.)^* \mid \alpha_y\overline{\alpha_k}.)^*)\backslash\{\alpha_y\} = ((\alpha_x\overline{\alpha_j}\overline{\alpha_k}.)^*)\backslash\{\alpha_y\}$ 6 & recursive application of Theorem 25

8: $(\alpha_x\overline{\alpha_j}\overline{\alpha_k}.)^* \perp \{\alpha_y\}$ Uniqueness of α

9: $((\alpha_x\overline{\alpha_j}\overline{\alpha_y}.)^* \mid (\alpha_y\overline{\alpha_k}.)^*)\backslash\{\alpha_y\} = (\alpha_x\overline{\alpha_j}\overline{\alpha_k}.)^*$

10: $\langle \mathbf{M}_j \mid \langle \mathbf{M}_k \rangle\rangle = ((\alpha_x\overline{\alpha_j}\overline{\alpha_k}.)^* \mid \mathbf{M}_j \mid \mathbf{M}_k \mid ((\overline{\iota_k}\iota_y.)^* \mid (\overline{\iota_j}\overline{\iota_y}\iota_x.)^*)\backslash\{\iota_y\})\backslash\{\alpha_k,\iota_k,\alpha_j,\iota_j\}$

5 & 9

11: $((\overline{\iota_k}\iota_y.)^* \mid (\overline{\iota_j}\overline{\iota_y}\iota_x.)^*)\backslash\{\iota_y\} = (\overline{\iota_k}\overline{\iota_j}\iota_x.)^*$ Similar to step 6-9

12: $\langle \mathbf{M}_j \mid \langle \mathbf{M}_k \rangle\rangle = ((\alpha_x\overline{\alpha_j}\overline{\alpha_k}.)^* \mid \mathbf{M}_j \mid \mathbf{M}_k \mid (\overline{\iota_k}\overline{\iota_j}\iota_x.)^*)\backslash\{\alpha_k,\iota_k,\alpha_j,\iota_j\}$ 10 & 11

13: $\langle \mathbf{M}_j \mid \langle \mathbf{M}_k \rangle\rangle = \langle \mathbf{M}_j \mid \mathbf{M}_k \rangle$ 12 & Definition of $\langle\rangle$

Q.E.D.

B.1 Parallelization Theorem

The most important theorem of our algebra is the ***parallelization theorem*** which was given as Theorem 4 in section 3.5.1 on page 63:

$$\mathbf{M}_a \perp \mathbf{M}_b, \mathbf{M}_a \perp \mathbf{M}_r, \mathbf{M}_b \perp \mathbf{M}_q, \mathbf{M}_q \perp \mathbf{M}_r \Rightarrow$$

$$\langle \mathbf{M}_a \mid \mathbf{M}_b \mid \mathbf{M}_p; \mathbf{M}_q ; \mathbf{M}_r ; \mathbf{M}_s \rangle \backslash S =$$

$$\langle \mathbf{M}_a \mid \mathbf{M}_b \mid \mathbf{M}_p; \langle \mathbf{M}_q \mid \mathbf{M}_r \rangle; \mathbf{M}_s \rangle \backslash S =$$

$$\langle \mathbf{M}_a \mid \mathbf{M}_b \mid \mathbf{M}_p; \mathbf{M}_r ; \mathbf{M}_q ; \mathbf{M}_s \rangle \backslash S$$

where $S = Sort(\mathbf{M}_a) \cup \overline{Sort}(\mathbf{M}_a) \cup Sort(\mathbf{M}_b) \cup \overline{Sort}(\mathbf{M}_b) \cup Sort(\mathbf{M}_q) \cup \overline{Sort}(\mathbf{M}_q) \cup Sort(\mathbf{M}_r) \cup \overline{Sort}(\mathbf{M}_r)$. The proof for this theorem is rather complicated. We will first outline the proof approach, then give the theorem proof assuming that the needed lemmas are true, and finally prove the lemmas.

The heart of the proof is to show that no matter what states $\mathbf{M}_a$ and $\mathbf{M}_b$ are in when $\mathbf{M}_p$ goes idle (call these states A and B), if $\mathbf{M}_a$ and $\mathbf{M}_b$ can reach states A' and B' when both $\mathbf{M}_q$ and $\mathbf{M}_r$ have gone idle ($\mathbf{M}_s$ has just been activated) then $\mathbf{M}_a$ and $\mathbf{M}_b$ can reach those states regardless of whether they interact with $\mathbf{M}_q$; $\mathbf{M}_r$ or $\langle \mathbf{M}_q \times \mathbf{M}_r \rangle$ or $\mathbf{M}_r$; $\mathbf{M}_q$. Next, we show that once $\mathbf{M}_s$ has been activated only the states of machines $\mathbf{M}_a$ and $\mathbf{M}_b$ at the beginning of the interaction can influence the subsequent behavior of the composition.

Our proof will be based upon the definition of failures equivalence. Recall that failures equivalence says that:

1 each machine has a trace *t* if and only if the equivalent machine also has a trace *t*, and

2 each machine can reject a set S after trace *t* if and only if the equivalent machine also can reject a set S after trace *t*.

For the sake of discussion, we will call the initial states of these three configurations X, Y and Z, with:

$$X \equiv \langle \mathbf{M}_a \mid \mathbf{M}_b \mid \mathbf{M}_p ; \langle \mathbf{M}_q ; \mathbf{M}_r \rangle ; \mathbf{M}_s \rangle \backslash S$$

$$Y \equiv \langle \mathbf{M}_a \mid \mathbf{M}_b \mid \mathbf{M}_p ; \langle \mathbf{M}_q \mid \mathbf{M}_r \rangle ; \mathbf{M}_s \rangle \backslash S$$

$$Z \equiv \langle \mathbf{M}_a \mid \mathbf{M}_b \mid \mathbf{M}_p ; \langle \mathbf{M}_r ; \mathbf{M}_q \rangle ; \mathbf{M}_s \rangle \backslash S$$

We begin with a simple lemma that establishes that the failures of a state with no observable actions of its own are simply the union of the failures of its successors.

Lemma 28

$$\forall a \neq 1 : \in Act \neg \exists S' \text{ s.t. } S \xrightarrow{a} S' \Rightarrow$$

$$Failures(S) = \{(s,A) \mid S \xrightarrow{1} S' \text{ and } (s,A) \in Failures(S')\}$$

Proof: We begin by recalling the definition of failures and initials:

$$Failures(S) = \{(s,A) \mid \exists S' \text{ s.t. } S \stackrel{s}{\Rightarrow} S' \text{ and } A \cap Intials(S') = \varnothing\}$$

$$Intials(T) = \{a : \in Act \mid \exists T' \text{ s.t. } T \stackrel{\langle a \rangle}{\Rightarrow} T'\}$$

We note that if S has no transitions other than 1 then for all successors S', *Intials*(S) = $\cup_{S'}$*Intials*(S'), *Intials*(S') $\subseteq$ *Intials*(S), and therefore *A-Intials*(S)$\subseteq$*A-Intials*(S'). Therefore every failure of S is also a failure of S'.

We now prove lemma about the behavior of the three machines between the time that M_p goes idle and M_s is activated.

Lemma 29

Let A be a possible state of M_a at the time that M_p goes idle, and similarly let B be a possible state of M_b at the time that M_p goes idle. Now let Let A' be a possible state of M_a at the time that M_s goes active, and similarly let B' be a possible state of M_B at the time that M_a goes active. Then:

$$A|B|\langle M_q;M_r\rangle \overset{\overline{\alpha_c},\overline{\iota_c}}{\Rightarrow} A'|B' \tag{70}$$

$$A|B|\langle M_q \mid M_r\rangle \overset{\overline{\alpha_c},\overline{\iota_c}}{\Rightarrow} A'|B' \tag{71}$$

$$A|B|\langle M_r;M_q\rangle \overset{\overline{\alpha_c},\overline{\iota_c}}{\Rightarrow} A'|B' \tag{72}$$

where the term α_c is the activation action for the term in braces, and ι_c is the idle action for this term.

Proof: Expanding the terms we have, for X, the following transition as M_p goes idle:

$$A|B|(\alpha_c\overline{\alpha_q}.)^*|\alpha_q.M_q|(\overline{\iota_q}\overline{\alpha_r}.)^*|\alpha_r.M_r|(\iota_c\overline{\iota_r}.)^* \overset{\overline{\alpha_c}}{\Rightarrow} A|B|\mathbf{0}|M_q|(\overline{\iota_q}\overline{\alpha_r}.)^*|\alpha_r.M_r|(\iota_c\overline{\iota_r}.)^* \tag{73}$$

where M_q is the first state of $\mathbf{M}_q$ after activation and M_r is the first state of $\mathbf{M}_r$ after activation. Strictly speaking, the third term on the right should still be $(\alpha_c\overline{\alpha}_q.)^*$, but since α_c can never happen again until the entire machine (X) is started again, we note that it is behaviorally equivalent to **0** and use **0** to indicate this.

Now M_q is free to interact with A, but may not interact with any of the other terms due to orthogonality until it goes idle, at which time an interaction with $(\bar{\iota}_q\overline{\alpha}_r.)^*$ will occur. We also note that B will not change state by interacting with other terms since it is orthogonal to all but $\alpha_r.M_r$, and interactions with this term cannot occur until after α_r has occurred.

Interactions between M_q and A can continue until M_q goes idle, at which point we have:

$$A|B|\mathbf{0}|M_q|(\bar{\iota}_q\overline{\alpha}_r.)^*|\alpha_r.M_r|(\iota_c\bar{\iota}_r.)^* \Rightarrow A'|B|\mathbf{0}|\mathbf{0}|\mathbf{0}|M_r|(\iota_c\bar{\iota}_r.)^* \quad (74)$$

again using **0** to represent terms that are inactive. Note at this point that A has transitioned to A', and that A' is orthogonal to the remaining terms. Therefore A' is already in its final form (we assume, without loss of generality, that A' has already completed whatever silent transitions that it is going to make). Now M_r and B are free to interact until M_r goes idle, at which point B has transformed to its final state B'. Thus we have:

$$A'|B|\mathbf{0}|\mathbf{0}|\mathbf{0}|M_r|(\iota_c\bar{\iota}_r.)^* \stackrel{\bar{\iota}_c}{\Rightarrow} A'|B'|\mathbf{0}|\mathbf{0}|\mathbf{0}|\mathbf{0}|\mathbf{0} \quad (75)$$

Putting (73), (74) and (75) together, we get (70), which is the desired result for X. The derivation for (72) is similar. We now turn to the derivation of (71).

Expanding the terms we have, for Y, the following transition as $\mathbf{M}_p$ goes idle:

$$A|B|(\alpha_c\overline{\alpha}_q\overline{\alpha}_r.)^*|\alpha_q.M_q|\alpha_r.M_r|(\iota_c\overline{\iota}_q\overline{\iota}_r.)^* \stackrel{\overline{\alpha}_c}{\Rightarrow} A|B|\mathbf{0}|M_q|M_r|(\iota_c\overline{\iota}_q\overline{\iota}_r.)^* \quad (76)$$

where M_q is the first state of $\mathbf{M}_q$ after activation and M_r is the first state of $\mathbf{M}_r$ after activation. Now M_q is free to interact with A, but M_q and A may not interact with any of the other terms due to orthogonality until M_q goes idle, at which time an interaction with $(\iota_c\overline{\iota}_q\overline{\iota}_r.)^*$ will occur. Similarly, M_r and B are free to interact with each other, but M_r and B may not interact with any of the other terms due to orthogonality until M_r goes idle, at which time an interaction with $(\iota_c\overline{\iota}_q\overline{\iota}_r.)^*$ will occur. At this point we have:

$$A|B|\mathbf{0}|M_q|M_r|(\iota_c\overline{\iota}_q\overline{\iota}_r.)^* \stackrel{\overline{\iota}_c}{\Rightarrow} A'|B'|\mathbf{0}|\mathbf{0}|\mathbf{0}|\mathbf{0}|\mathbf{0} \quad (77)$$

Putting (73) and (77) together, we get (71), which is the desired result for Y.

We conclude by observing that in each case the sole determining factors for the final state A' was the initial state A and the machine $\mathbf{M}_q$. Similarly, the final state B' was determined entirely from the initial state B and the machine $\mathbf{M}_r$.We thus conclude that our lemma is, indeed, valid.

Proof of Theorem 4

We show that the *Failures*(X) = *Failures*(Y) = *Failures*(Z). To show this, we must show that the possible traces s are the same for all three machines, and that after each trace s the set of failures (sets of pairs of the form (s,A)).

Considering the set of traces first, we note that for traces involving just the actions of $\mathbf{M}_p$, the behavior of all three machines is determined entirely by $\mathbf{M}_a \mid \mathbf{M}_b \mid \mathbf{M}_p$ since the other

machines have not been activated yet. Thus the traces up to the time that M_p goes idle are identical for all three machines, and therefore all three machines can reach the same set of $A|B$ states for $M_a \mid M_b$. We then note that by Lemma 29 that the set of states $A'|B'$ reachable from $A|B$ are the same for all three machines (note that no observable actions have occurred). We further note that after M_s is activated, the behavior of the machine is determined entirely by $A'|B'|M_s$, since the other machines are now idle. We therefore conclude that the set of possible traces for all three machines are the same.

Now we turn to the consideration of failures. Let S' be a state that we have reached via the sequence s. We consider three cases:

Case 1: S' is a state after M_s is activated. Then the same state is reachable in all three machines (established in the trace argument). Since the failures are determined entirely by subsequent behavior, exactly the same failures are possible for all three machines.

Case 2: S' is a state after M_p goes idle and before M_s is activated. By Lemma 28, the failures of this state are subsumed by the union of the failures of the $A'|B'$ states that are possible when M_s is activated. Furthermore, in order to reach S', we had to go through an $A|B$ state in which M_p went idle. The failures of the $A|B$ state are also failures of the machine after s. By Lemma 28, the failures of the $A|B$ state subsume the failures of the S' state. Lemma 28 also tells us that the failures of the $A|B$ state are the union of the failures of all of the $A'|B'$ states that can be reached from the $A|B$ state. Lemma 29 tells us that the same set of $A'|B'$ states are reachable in all three machines, and therefore the failures of all three machines for this $A|B$ state are the same. Since the failures of S' are subsumed by the $A|B$ state, we conclude that its failures are also failures of the other two machines.

Case 3: S' is a state before $\mathbf{M}_p$ goes idle. From the definition of failures, we see that the failures of the state are determined by the initials of the state. Now the initials are either actions of $\mathbf{M}_p$ or $\mathbf{M}_s$. If the initial is an action of $\mathbf{M}_p$, then it is a possible initial for all three machines since the operation of the machines prior to idling $\mathbf{M}_p$ is identical for all three machines. If the action is an action of $\mathbf{M}_s$, then there had to be an $A|B$ state and an $A'|B'$ on the path from S' to the state in $\mathbf{M}_s$ reached by the action. By Lemma 29 this same transition is possible in the other machines as well, and therefore the same action is an initial of this state for all three machines. Since the initials of $S'$are the same in all three cases, the failures of S' are the same.

Lemma 30 Unique Factorization of Actions

Here we show that factorization of an action into orthogonal actions belonging to the same sort is unique up to =. Let S_a ands S_b be two sorts, with $S_a \perp S_b$. Let $a_1, a_2, b_1, b_2, \gamma$ be actions, with $a_1 b_1 = \gamma = a_2 b_2$, $Atom(a_1) \subseteq S_a$, $Atom(a_2) \subseteq S_a$, $Atom(b_1) \subseteq S_b$, $Atom(b_2) \subseteq S_b$. Then we have:

$$a_1 b_1 = \gamma = a_2 b_2 \text{ implies } a_1 = a_2 \text{ and } b_1 = b_2$$

1: Since $(Act, \times, 1)$ is an Abelian group (in particular, it is commutative and associative), we can define a canonical form for each element $\gamma \in Act$ as a product of primitive actions, each taken to some power: $g_1^i \times g_2^j \times g_3^k \times \ldots$. If γ_1 and γ_2 are both represented by the same canonical form, then $\gamma_1 = \gamma_2$.

2: Let us consider the canonical form of γ. Since $S_a \perp S_b$, each primitive action (except for 1) belongs to either S_a or S_b. The action 1 belongs to both, and without loss of

generality, we shall assume that 1 is a member of every action. Now for each primitive action of γ (except 1), if the action belongs to S_a, the it must appear in both $Atom(a_1)$ and $Atom(a_2)$, since, by definition, it cannot be in S_b, and therefore cannot be in $Atom(b_1)$ or $Atom(b_2)$. Since these are the only primitive actions in a_1 and a_2, we have $Atom(a_1) = Atom(a_2)$. Similar arguments can be use to establish that $Atom(b_1) = Atom(b_2)$.

3: Let us again consider the primitive actions of γ, this time considering the number of times that each atom occurs. Since $a_1b_1 = \gamma = a_2b_2$, then if atom g_i ($g_i \neq 1$) occurs n times in γ, it must occur n times in a_1b_1 and a_2b_2 as well. If $g_i \in S_a$ then $g_i \in Atom(a_1)$ and $g_i \in Atom(a_2)$ and occurs in a_1 and a_2 exactly n times. Similarly, if $g_i \in S_b$ then $g_i \in Atom(b_1)$ and $g_i \in Atom(b_2)$ and occurs in b_1 and b_2 exactly n times. Thus both a_1 and a_2 have the same canonical form, and b_1 and b_2 have the same canonical form. Thus $a_1 = a_2$ and $b_1 = b_2$. Q.E.D.

Lemma 31 Unique Factorization of Agents

If $E_1 \perp F_1$, $Sort(E_1)=Sort(E_2)$, $Sort(F_1)=Sort(F_2)$ then

$E_1|F_1=E_2|F_2$ implies $E_1=E_2$ and $F_1=F_2$

Lemma 32 Interleaving Lemma

$A \perp B$ implies $a \in Traces(A)$ and $b \in Traces(B)$ iff $\forall c \in \text{Comp}(a,b)$ $c \in Traces(A|B)$

In the forward direction this is just the definition of *Traces*(A|B). In the reverse direction we simply observe that because A⊥B the factorization of *c* into orthogonal components is Application of Theorems and Corollaries

B.2 Proof for Change of Scope Theorem

The proof of Theorem 5 is relatively simple. This theorem states:

$$\mathbf{M}_a \perp \mathbf{M}_x,\ \mathbf{M}_c \perp \mathbf{M}_x \Longrightarrow$$

$$\langle \mathbf{M}_a ; \langle \mathbf{M}_b \mid \mathbf{M}_x \rangle ;\ \mathbf{M}_c \rangle \backslash S = \langle \mathbf{M}_x \mid \mathbf{M}_a ; \mathbf{M}_b ;\ \mathbf{M}_c \rangle \backslash S$$

where $S = Sort(\mathbf{M}_X)$.

Since $\mathbf{M}_x$ cannot interact with either $\mathbf{M}_a$ or $\mathbf{M}_c$, and the actions of $\mathbf{M}_x$ are hidden, then the traces of both configurations are the same.

Since all of the actions of $\mathbf{M}_x$ are hidden, then the actions of $\mathbf{M}_x$ are part of the refusal set for all states of both configurations. Thus the fact that $\mathbf{M}_x$ is actually in a different state in the two configurations during the time that $\mathbf{M}_a$ and $\mathbf{M}_c$ are operating does not affect the refusal sets.

Since the traces are the same and the refusals are the same, the two configurations are failures equivalent.

C Masking Union Properties and Visibility Proofs

C.1 Masking Union Properties

The proofs of the following properties follow simply from the definition of the masking union, and are not given. In the following, the homograph relations ***H***(a,b) is assumed to be any arbitrary relation.

Idempotence:

$$A \underset{m}{\cup} A = A \tag{78}$$

Left Identity:

$$\varnothing \underset{m}{\cup} A = A \tag{79}$$

Right Identity:

$$A \underset{m}{\cup} \varnothing = A \tag{80}$$

Masking union is distributive over union:

$$A \underset{m}{\cup} (B \cup C) = (A \underset{m}{\cup} B) \cup (A \underset{m}{\cup} C) \tag{81}$$

Union is *not distributive* over masking union. Consider the following expression, and an element in A that has a homograph in C. The expression on the left would result in both elements being in the result, while the expression on the right would only have the element from A in the result.

$$A \cup (B \underset{m}{\cup} C) \neq (A \cup B) \underset{m}{\cup} (A \cup C)$$

The masking union is *not commutative*.

$$(A \underset{m}{\cup} B) \neq (B \underset{m}{\cup} A)$$

The masking union is associative if and only if the homograph relation being used is transitive.

$$(A \underset{m}{\cup} B) \underset{m}{\cup} C \neq A \underset{m}{\cup} (B \underset{m}{\cup} C)$$

Subsumption: Given: $X \subseteq A$

$$A \underset{m}{\cup} X = A \tag{82}$$

The converse is not true:

$$X \underset{m}{\cup} A \neq A$$

$$A \underset{m}{\cup} B = A \underset{m}{\cup} (X \cup B) \tag{83}$$

Additional properties:

$$(A \underset{m}{\cup} B) \cup A = A \cup (A \underset{m}{\cup} B) = A \underset{m}{\cup} B \tag{84}$$

$$(A \underset{m}{\cup} B) \underset{m}{\cup} (A \underset{m}{\cup} C) = A \underset{m}{\cup} (B \underset{m}{\cup} C) = (A \underset{m}{\cup} B) \underset{m}{\cup} C \tag{85}$$

$$(A \cup B) \underset{m}{\cup} B = A \cup B \tag{86}$$

$$(A \cup B) \underset{m}{\cup} (A \cup C) = (A \cup B) \underset{m}{\cup} C \tag{87}$$

Theorem 33 Computation of Direct Environments

Given:

a an ordered set **LD** of k declarations $\{d_1, d_2, \ldots d_k\}$

b an initial direct environment $\mathbf{DE}_0$

c an empty set $\mathbf{LD}_0$

We define a partial local declaration set $\mathbf{LD}_i$, as follows:

$$\mathbf{LD}_i = \{d_i\} \cup \mathbf{LD}_{i-1}$$

and the direct environment associated with each declaration:

$$\mathbf{DE}_i = \mathbf{LD}_i \underset{m}{\cup} \mathbf{DE}_{i-1}$$

Then we claim that

$$\mathbf{DE}_k = \mathbf{LD}_k \underset{m}{\cup} \mathbf{DE}_0$$

Proof by Induction on i:

Basis: i = 1

1: $\mathbf{DE}_1 = \mathbf{LD}_1 \underset{m}{\cup} \mathbf{DE}_0$ Definition of $\mathbf{DE}_i$

Inductive Step:

1: $\mathbf{DE}_i = \mathbf{LD}_i \underset{m}{\cup} \mathbf{LD}_{i-1} \underset{m}{\cup} \mathbf{DE}_0$ by inductive hypothesis

2: $\mathbf{DE}_i = \mathbf{LD}_i \overset{\cup}{m} \mathbf{DE}_0$ 1, Eq. (82) & Def'n of $\mathbf{LD}_i$

Q.E.D.

References

[Aho74] Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffrey D., The Design and Analysis of Computer Algorithms, Addison-Wesley (1974)

[Amad86] Amadio, Roberto, Bruce, Kim B., and Longo, Giuseppe, "The Finitary Projection Model for Second Order Lambda Calculus and Solutions to Higher Order Domain Equations," in Proceedings, 1986 Symposium on Logic in Computer Science, document order number 720, IEEE Computer Society Press order number 720, 1730 Massachusetts Avenue N.W., Washington D.C. 20036-1903 [1986]

[Bjor80] Bjorner, D. and Oest, O.N., Eds, Towards a Formal Description of Ada, LNCS Vol 98, Springer-Verlag (1980)

[Broo83] Brookes, Stephen D, A Model for Communicating Sequential Processes, Ph.D. thesis, University College, Oxford University (1983)

[Brow89] Brown, Paul C., Oconnor, D.M., and Kelliher, Tim, "An Extended Overload Resolution Algorithm that allows Types and Subprograms as First Class Objects," internal document, GE Corporate Research and Development Center, Schenectady, New York (1989)

[Brow90] Brown, Paul C., Computing Visibility in Programming Languages, Technical Report 90CRD098, GE Research and Development Center, Schenectady, New York 12301 (1990)

[Broy87] Broy, Manfred and Wirsing, Martin, "On the Algebraic Definition of Programming Languages," ACM TOPLAS Vol. 9., No. 1, (1987)

[Bruc84] Bruce, Kim B., and Meyer, Albert R., "The Semantics of Second Order Polymorphic Lambda Calculus," in Semantics of Data Types (Kahn, MacQueen and Plotkin, editors) pp 131-144, Lecture Notes in Computer Science, vol 173, Springer Verlag [1984]

[Bruc87] Bruce, Kim B., Meyer, Albert R., and Mitchell, John C., "The Semantics of Second Order Lambda Calculus," preprint (1987)

[Eaker91a] Eaker, Charles E., "Creating Software Should Be Easy," (unpublished) GE Corporate Research and Development Center, Schenectady, New York (1991)

[Eaker91b] Eaker, Charles E., "How to Create Software: A Guide to the Perplexed," (unpublished) GE Corporate Research and Development Center, Schenectady, New York (1991)

[Ende73] Enderton, Herbert B., A Mathematical Introduction to Logic, Academic Press (1973)

[Good86] Goodenough, John B. The Ada Compiler Validation Capability Implementor's Guide, Version 1, SofTech, Inc., Waltham, Ma. 02254-9197 (1986)

[Gord79] Gordon, Michael J.C., The Denotational Description of Programming Languages, Springer-Verlag (1979)

[Gogu77] Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B., "Initial Algebra Semantics and Continuous Algebras," JACM Vol. 24, No. 1 (1977)

[Hoar69] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," Communications of the ACM, Vol. 12, No. 10 (1969)

[Hoar85] Hoare, C.A.R., Communicating Sequential Processes, Prentice Hall International (1985)

[LRM] Ada Programming Language, ANSI/MIL-STD-1815A (1983)

[Mann74] Manna, Zohar, Mathematical Theory of Computation, McGraw Hill (1974)

[McCr82] McCracken, Nancy, "A Finitary Retract Model for the Polymorphic Lambda Calculus," Technical Report 83-2, Syracuse University [1982]

[McN82] McNaughton, Robert, Elementary Computability, Formal Languages and Automata, Prentice-Hall (1982)

[Mend87] Mendelson, Elliott, Introduction to Mathematical Logic,Wadsworth & Brooks/Cole (1987)

[Miln80] Milner, Robin, A Calculus of Communicating Systems, LNCS Vol 92, Springer-Verlag, (1980)

[Miln83] Milner, Robin, "Calculi for Synchrony and Asynchrony," Journal of Theoretical Computer Science, Vol. 25, pp 267-310 (1983)

[Miln89] Milner, Robin, Communication and Concurrency, Prentice Hall International (1989)

[Mitc84] Mitchell, John C., "Semantic Models for Second-Order Lambda Calculus," in Proceedings of the 25th Annual Symposium on Foundations of Computer Science, IEEE document 84CH2085-9 (1984)

[Modu88] Modula-2 Working Group, "A Formal Definition of Modula-2," draft version dated 30 June, 1988

[Moss83] Mosses, Peter, "Abstract Semantic Algebras!," in Formal Descriptions of Programming Concepts II, D. Bjørner (ed), North-Holland (1983)

[Moss84] Mosses, Peter, "A Basic Abstract Semantic Algebra," in Semantics of Data Types, LNCS Vol 173 (1984)

[Rees86] Rees, Jonathan and Clinger, William (eds), "Revised^3 Report on the Algorithmic Language Scheme," SIGPLAN Notices, Vol. 21, No. 12 (1986).

[Scot76] Scott, Dana, "Data Types at Lattices," in *SIAM Journal of Computing*, Vol 5, No. 3 (1976)

[Scot82] Scott, Dana S., "Domains for Denotational Semantics," LNCS Vol 140, Springer-Verlag (1982)

[Stee84] Steele, Guy L., Common Lisp, The Language, Digital Press (1984)

[Stoy77] Stoy, Joseph E., Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press (1977)

[Wirt82] Wirth, Niklaus, Programming in Modula-2, Springer-Verlag (1982)